Clube Delphi + PHP

DevMedia group

Ano 9 · Edição 107 · R$11,90

DataSnap 2009

Acerele suas aplicações multicamadas com o poderoso driver DataSnap

Ask th Expert
Detectando unidades removíveis

QuickUpdate
Google Maps: a API em detalhes

Banco de Dados
Veja algumas dicas para desenvolvimento de aplicações de migração

EasyDelphi
Criando aplicações com histórico em XML

Expert
Saiba como enviar e receber dados das aplicações Cliente e Servidora

PHP
Construa WebServices em PHP com o Delphi for PHP

PHP
Introdução a POO com PHP

ISSN 1517990-7

9 771517 990009 00107

Na Web

Confira no portal ClubeDelphi PLUS 2 vídeo-aulas:

- Novidades do RAD Studio 2009 – Partes VIII e IX

SISFÁCIL 2.0

FICOU AINDA, MAIS FÁCIL ADMINISTRAR SEU SITE

Enquete, Fotos, Notícias, Blogs e muito mais.

DISPONÍVEL PARA CLIENTES RJHOST

Seu site pronto e completo

www.sisfacil.com.br

RJHOST

Sumário

Olá, eu sou o **DevMan**! Desta página em diante, eu estarei lhe ajudando a compreender com ainda mais facilidade o conteúdo desta edição. Será um prazer contar com sua companhia! Confira abaixo o que teremos nesta revista:

Delphi – Banco de Dados

12 – Migrando dados de um banco para outro
Veja algumas dicas para desenvolvimento de aplicações de migração
[*Matheus Oliveira Santos*]

Delphi – Expert

26 – Transporte de dados em multicamadas e DataSnap
Saiba como enviar e receber dados das aplicações Cliente e Servidora
[*Daniel Sobrinho*]

Boas Práticas – Boas Práticas

34 – DataSnap 2009
Conheça as principais novidades dessa tecnologia para desenvolvimento multicamadas
[*Fabricio Hissao Kawata, Adriano Santos e Daniel Sobrinho*]

Easy Delphi – Easy Delphi

45 – Criando aplicações com histórico em XML
Saiba como reaproveitar as últimas consultas do seu banco de dados
[*Daniel Sobrinho*]

PHP – PHP

55 – Web Services no Delphi for PHP
Compartilhe serviços via web e integre sua aplicação com clientes e fornecedores
[*Rodrigo Carreiro*]

PHP – PHP

63 – Introdução a orientação a objetos com PHP
Conheça os princípios básicos da POO com PHP
[*Rafael Silva*]

Você percebeu os ícones ao lado de cada matéria? Eles indicam o que você vai encontrar no artigo – dessa forma, você também pode ter uma idéia geral do que vai encontrar nesta edição como um todo! Os editores trabalham sempre no sentido de fechar a revista seguindo esta definição, para oferecer a você o melhor conteúdo didático!

Confira abaixo a lista com a definição dos tipos de artigo encontrados nesta edição:

[Boas Práticas] Um dos objetivos da revista é levar para o leitor não somente as melhores técnicas mas também as melhores práticas. Esse tipo de artigo foca em técnicas que poderão aumentar a qualidade do desenvolvimento de software.

[Boa Idéia] A esse tipo de artigo é quase um artigo "Mão na Massa" - a diferença é que o exemplo do artigo é tão criativo que é considerado uma "Boa Idéia", dentro do contexto do desenvolvimento de software.

[Expert] Artigo com foco no leitor avançado.

[Novidades] Artigos sobre tecnologias de ponta, que ainda não fazem parte o dia-a-dia do desenvolvedor mas que prometem ser a próxima febre.

[Easy Delphi] Com foco no desenvolvedor iniciante.

[PHP] Artigo com foco no PHP puro.

Clube Delphi +PHP

Ano 9 - 107ª Edição 2009 - ISSN 1517990-7 Impresso no Brasil

Corpo Editorial

Editor Geral
Adriano Santos
adriano.santos@devmedia.com.br

Editor Técnico
Paulo Quicoli
pauloquicoli@devmedia.com.br

Daniel Sobrinho
daniel.sobrinho@devmedia.com.br

Comissão Editorial
Fabrício Desbessel, Paulo Quicoli, Adriano Santos e Guinther Pauli.

Diagramação
Romulo Araujo
romulo@devmedia.com.br

Coordenação Geral
Daniella Costa - daniella@devmedia.com.br

Revisão e Supervisão
Thiago Vincenzo - thiago.v.ciancio@devmedia.com.br

Distribuição
Fernando Chinaglia Dist. S/A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

Carmelita Mulin – Atendimento ao Leitor
www.devmedia.com.br/central/default.asp
(21) 3382-5025

Kaline Dolabella – Gerente de Marketing e Atendimento
kalined@terra.com.br
(21) 3382-5025

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

Kaline Dolabella
publicidade@devmedia.com.br

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!
Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Adriano Santos - Editor da Revista
adriano.santos@devmedia.com.br

EDITORIAL

Olá a todos. Desde que o DataSnap foi introduzido no Delphi lá atrás, já se esperava muito da tecnologia. Passamos por inúmeras modificações, implementações, melhorias no código, introdução de componentes, enfim. O pessoal da Borland, depois CodeGear, não brincaram em serviço e foram melhorando cada vez mais a tecnologia. Mas creio que dessa vez se superaram. Um novíssimo DataSnap foi introduzido na versão 2009 do Delphi, o DataSnap 2009 como está sendo chamado. Uma das principais novidades é a extinção do uso de objetos COM e introdução do driver DataSnap. Sem dúvida nenhuma uma das mais importantes melhorias já feitas até hoje. Velocidade, praticidade e facilidade de desenvolvimento são apenas algumas das vantagens. Nessa edição fiz questão de olhar de perto tudo isso e publicar um artigo juntamente com o colunista Fabricio Hissao Kawata e com nosso mais novo editor técnico Daniel Sobrinho. Juntos nós estudamos o DataSnap 2009 e montamos um artigo bastante completo cobrindo tudo que há de novo. Confira você mesmo.

O Daniel ainda continua na revista mostrando o todo poder do DataSnap em uma aplicação multicamadas onde ele mostra como fazer o transporte de dados entre cliente e servidor. Confira também seu artigo que mostra como criar histórico de consultas em formato XML. Isso agiliza o processo de busca de informações no servidor, sensacional.

O Matheus fala sobre sua experiência no desenvolvimento de aplicações para migração de dados. Sem dúvida uma das partes mais chatas para um desenvolvedor. Migrar dados de clientes não é uma tarefa simples.

Em PHP temos dois excelentes artigos. O primeiro, de Rodrigo Carreiro, fala sobre o desenvolvimento de Web Services com Delphi for PHP. Veremos com ele que é bem simples trabalhar com Web Services. O segundo artigo é do colunista Rafael Silva que fala específicamente de POO com PHP puro. Veja os principais conceitos e como desenvolver usando POO e PHP.

Uma boa leitura, muito sucesso a todos e até a próxima.

Adriano Santos

adriano.santos@devmedia.com.br

Editor Chefe da revista Clube Delphi
Conheça nosso BLOG em *clubedelphiphp.blogspot.com*

ClubeDelphi PLUS

DevMedia group

A revista ClubeDelphi é parte integrante da assinatura ClubeDelphi PLUS.
Para mais informações sobre o pacote PLUS, acesse:
http://www.devmedia.com.br/clubedelphi/portal.asp

Hospede seus sites de forma simples, rápida e ilimitada.

Plano **Básico**

- 1 domínio principal
- 50 domínios hospedados
- Transferência **ilimitada**
- Espaço **ilimitado**
- 10 contas de e-mail
- PHP, ASP e ASP.NET
- Painel de controle Plesk
- Suporte 24hs

Tudo isso por apenas

R$ **15,90** por mês
pagamento trimestral

Assine agora mesmo e descubra por que Hospedix é a hospedagem de sites preferida entre os profissionais.

- Conheça também nossos planos profissionais com registro de domínio grátis e isenção de taxas anuais enquanto for cliente.

HOSPEDIX
HOSPEDAGEM PROFISSIONAL

www.hospedix.com.br

30 DIAS GRÁTIS

Ao assinar, digite o código de desconto exclusivo para leitores desta revista e ganhe 30 dias inteiramente grátis.

RVSTMEDIA

ClubeDelphi PLUS

Informativo ClubeDelphi

Portal ClubeDelphi

www.clubedelphi.net/portal

+ 600 vídeo aulas e 7 cursos online

Brinde na web desta edição - 2 vídeos! Confira:

2 Vídeos

• **Novidades no Delphi 2009 - Parte VIII e IX**

O RAD Studio 2009 está repleto de novos controles e funcionalidades.
Confira todas as novidades nos vídeos de Rodrigo Carreiro. Nessa edição, grátis as aulas VIII e IX.

http://www.devmedia.com.br/articles/viewcomp.asp?comp=12197&hl=
http://www.devmedia.com.br/articles/viewcomp.asp?comp=12218

Para acessar os vídeos utilize os seguintes dados

Login: ***DVM.PLS*** *Senha:* ***dkl2305***

Gostou das vídeo aulas? O portal *www.devmedia.com.br* possui mais de **2 mil vídeo aulas** e **dezenas de cursos online** sobre desenvolvimento de software! Agora você pode comprar as vídeo aulas que preferir e fazer sua própria combinação de vídeos! Saiba mais em *www.devmedia.com.br/creditos*

Edições Anteriores da ClubeDelphi

Você pode comprar todas as edições anteriores da ClubeDelphi através do site da DevMedia!

Para isso basta acessar *https://seguro.devmedia.com.br/edicoes_anteriores.asp*

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Para votar, você vai precisar do código e senha de banca desta edição, que é:
Login: **DVM.PLS** Senha: **dkl2305**

Perguntas e Respostas

Identificando drivers removíveis

Perguntas respondidas por Paulo Quicoli
pauloquicoli@devmedia.com.br

O Heitor, nosso leitor, precisa de uma dica interessante. Em um sistema que ele desenvolve existe uma rotina para se copiar alguns arquivos, porém esses arquivos precisam ser copiados para um *pendrive*. Segundo ele, seus usuários apresentam dificuldades em utilizar um computador, não possuem *intimidade*. Então ele gostaria que de alguma forma ele pudesse exibir em seu sistema apenas os *drives* removíveis. Para solucionar esse caso, vamos utilizar uma *API* do *Windows* chamada *GetDriverType*. Essa função retorna o tipo de um determinado *drive* passado como parâmetro.

Vamos criar um procedimento que faça uso dessa *API* como código da **Listagem 1**.

Listagem 1. Procedimento RetornaDrivesRemoviveis

```
01 procedure TForm1.RetornaDrivesRemoviveis(Lista: TStrings);
02 var
03   Drive: Char;
04   s: TStringList;
05 begin
06   s := TStringList.Create;
07   for Drive := 'A' to 'Z' do
08     if GetDriveType(PChar(Drive+':\')) = DRIVE_REMOVABLE then
09       s.Add(Drive+':\');
10   Lista.AddStrings(s);
11   FreeAndNil(s);
12 end;
```

Basicamente estamos usando a função GetDriveType para retornar o tipo dos drives de A:\ à Z:\. Os drives que se enquadram no tipo DRIVE_REMOVABLE são adicionados à Lista, que é um parâmetro do tipo TStrings. Para utilizar esse procedimento podemos passar como parâmetro uma propriedade items de um ComboBox, como mostra o código a seguir.

```
RetornaDrivesRemoviveis(ComboBox1.Items);
```

Para testar, inseri um *pendrive* aqui em meu *notebook* e o resultado pode ser visto na **Figura 1**.

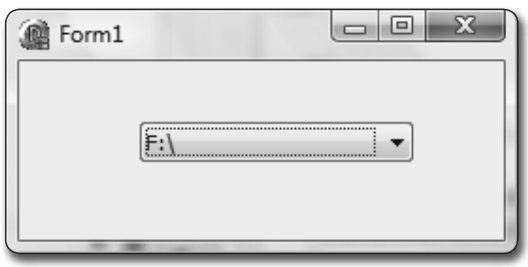


Figura 1. Drive F:\

Caso o computador de vocês possua um drive de disquete ele também aparecerá na lista. Isso porque o disquete também se enquadra no tipo *DRIVE_REMOVABLE*. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:
www.devmedia.com.br/clubedelphi/feedback

Quick Update: Google Maps e sua API

Veja mais alguns detalhes sobre essa API do Google

Adriano Santos
falecom@adrianosantos.pro.br
É desenvolvedor Delphi há mais de 10 anos, programador PHP, especialista em aplicações Win32 e API, bacharel em Comunicação Social. É Editor Chefe das revistas ClubeDelphi e WebMobile (Seção .net). Gerente de Desenvolvimento na SoftPark, membro fundador do DUG-SP (www.dug-sp.com) e mantenedor do blog Delphi to Delphi (www.delphitodelphi.blogspot.com).

Após a publicação do meu artigo sobre *Google Maps* na revista *ClubeDelphi 104*, recebi muitos e-mails de leitores informando que o código não funcionava, então resolvi investigar o problema para ver onde que ocorreu o erro. Descobri que o problema não está no código publicado na revista e sim por um motivo bem simples: a rua é nova e não consta no banco de dados do *Google* ou simplesmente há algo de errado em um dos endereços digitados. É o que diz a mensagem exibida pelo endereço original de exemplo utilizado na revista como podemos ver na **Figura 1**.

Ao traduzirmos a mensagem ao pé da letra, teremos o texto:

"A localização geográfica de um dos endereços não pode ser encontrada. Isso pode ser devido ao fato do endereço ser relativamente novo ou incorreto."

Muitos leitores conseguiram visualizar as respectivas rotas apenas modificando e testando outros endereços no exemplo criado. Em estudo descobri que podem ocorrer outros erros que na verdade são avisos, por exemplo. Quando joguei a rota:

• *Origem:* Rua Dança do Fogo, 91, Sao Paulo, SP;
• *Destino:* Rua Direita 100, Centro, Sao Paulo, SP;

A mensagem "Essa rota possui muitos vértices" aparece, como na **Figura 2**. Esse aviso aparece porque o caminho é bastante longo e um alerta é exibido. De qualquer forma a rota é carregada normalmente.

Uma alternativa para esse *erro* é omitir a mensagem que aparece apenas retirando-a do código *JavaScript* como vemos a em um trecho de código na **Listagem 1**.

Estudando mais ainda a *API* do *Google* percebi que houve uma pequena alteração nela. Quando jogamos um endereço que ele possivelmente não reconhece, o *Google Maps* mostra uma lista de possíveis ruas à esquerda do mapa. Experimente digitar o endereço: *Rua 15 de Novembro, 466, Sao Paulo, SP* no link *maps.google.com* e veja o que acontece (**Figura 3**). Uma lista é carregada a esquerda e o mais impressionante, nenhuma das ruas corresponde ao endereço digitado. A rua citada, uma das mais conhecidas

de São Paulo, não foi encontrada de forma alguma no *Google* para frustração do autor.

Testei pessoalmente diversas ruas que contém números como *Av 23 de Maio, Largo 13 de Maio, Rua 18 de Setembro*, e, aparentemente, o *Google* possui alguma dificuldade em encontrar as ruas ou a mesma está cadastrada de forma diferente, com algarismos romanos, por exemplo.

Então na realidade, o grande lance do *Google Maps* é estudar bem seus exemplos e tentar reproduzi-los na aplicação. O segredo do artigo foi transformar todo o conteúdo dos exemplos em uma grande *String*. As áreas variáveis foram concatenadas a essa grande *string* conforme exemplo na **Figura 4**.

Perceba que concatenamos a *string* que monta a *tag* em *HTML* ao texto que recebemos por parâmetro na função, veja mais um trecho de código:

```
procedure TForm1.CriarMapa(AOrigem, ADestino, ALocalidade: string);
[…]
Add('' + ''' + AOrigem + ''' + ',' + ''' + ADestino + ''' + ',' +
''' + ALocalidade + ''' + '); return false">');
[…]
```

Listagem 1. Trecho de código onde não há mais mensagem de aviso

```
[...]
  Add('      function onGDirectionsLoad(){');
  Add('        var poly = gdir.getPolyline();');
//  Add('        if (poly.getVertexCount() > 100) {');
//  Add('          alert("Esta rota possui muitos vértices.");');
//  Add('          return;');
//  Add('        }');
[...]
```

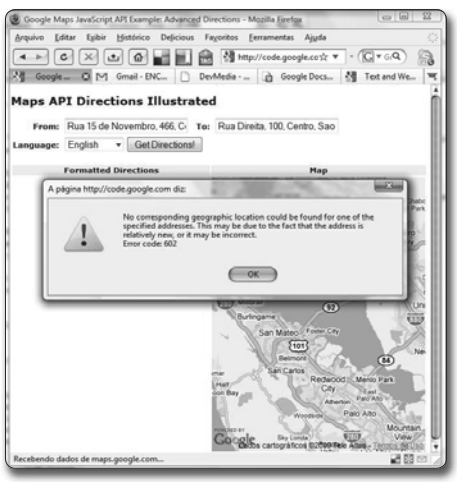


Figura 1. Erro na página original do Google

No final de tudo inserimos o conteúdo da *super-string* em uma variável do tipo *TStringList* e a salvamos no disco rígido como um arquivo *HTML*:

```
SaveToFile(ExtractFilePath(Application.ExeName)+'mapa2.html');
```

E então carregamos o arquivo no *TWebBrowser* para ser exibido para o usuário final. Realmente não há nenhum segredo.

```
WebBrowser1.Navigate(ExtractFilePath(Application.ExeName)+'mapa2.
html');
```

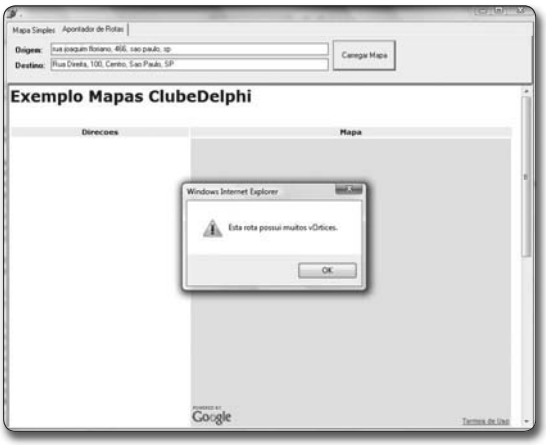


Figura 2. Mensagem de alerta

Figura 3. Sugestão de ruas

```
Add('  <form action="#" onsubmit="setDirections(' + ''' + AOrigem + ''' + ',' + ''' +
  ADestino + ''' + ',' + ''' + ALocalidade + ''' + '); return false">');
Add('   <input type="hidden" size="25" id="fromAddress" name="from" value="' + ''' +
  AOrigem + ''' + '"');
Add('   <input type="hidden" size="25" id="toAddress" name="to" value="' + ''' +
  ADestino + ''' + '"');
```

Figura 4. Trecho de código do Google Maps

O que recomendo mais uma vez é que se estude profundamente os exemplos oferecidos pelo *Google* para que então sejam aplicados em sistemas, tanto *Win32* quanto *Web*. Um forte abraço a todos e até a próxima.

Conclusão

A equipe do *Google* disponibiliza diversos serviços em sua grande maioria *API's* são disponibilizadas para que desenvolvedores façam uso delas. É importante o estudo sempre aprofundado para entender melhor como funcionam os serviços que a gigante oferece. Até a próxima. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

A partir de agora você pode contar com a ajuda da

Chegou a Consultoria On-line DevMedia

Consultoria Técnica + Professor Virtual + Certificação

Mais Informações:
www.devmedia.com.br/consultoria_online 21 3382-5025

DevMedia em seus projetos e estudos.

A DevMedia possui um numeroso time de autores, editores e professores que juntos produzem o material que você está acostumado a encontrar em nosso site e revistas. E são exatamente esses mesmos profissionais que estarão a sua disposição para tirar suas dúvidas e ajudá-lo em seus projetos e estudos. Através de uma plataforma 100% web a Consultoria DevMedia garante sigilo absoluto, eficiência e rapidez em todas as respostas. Finalmente você terá ao seu alcance uma consultoria de qualidade por um preço muito acessível. Consulte nossos planos.

Mais um serviço DevMedia group

Nesta seção você encontra artigos intermediários sobre Delphi Win 32 e Delphi for .NET

Migrando dados de um banco para outro

Veja algumas dicas para desenvolvimento de aplicações de migração

Nesse artigo veremos

- Criação de banco de dados Firebird;
- Exemplo de migração de dados.

Qual a finalidade

- Descobrir como recuperar os registros de um banco para outro com a estrutura das tabelas totalmente diferente;
- Exemplificar, de maneira prática, a solução para diferentes situações complicadas, independente do sistema gerenciador de banco de dados usado.

Quais situações utilizam esse recurso?

Quando se está implantando um novo sistema em um cliente e o mesmo precisa recuperar os dados já cadastrados em seu antigo sistema desenvolvido por terceiros a maior parte das vezes é criar uma aplicação para migração de dados, como fazer isso?

Resumo do DevMan

Como importar os dados de um banco para outro que possui a estrutura das tabelas totalmente diferente? Como inserir as dependências e manter a integridade dos dados? Uma aplicação em Delphi é a solução para a maioria dos problemas encontrados na hora de se importar os registros contidos em um banco de dados que será substituído. Este artigo traz um exercício prático para exemplificar a solução para situações como esta, a se aplicar independente do sistema gerenciador de banco de dados usado.

Matheus Oliveira Santos
matheusoliveira2@yahoo.com.br
É graduado em Ciência da Computação pela Universidade de Franca. Trabalha como programador em Delphi e banco de dados Firebird há mais de 3 anos.

Nada é mais importante para o usuário de um sistema do que manter o cadastro de seus clientes e outros dados importantes quando se fala em substituir seu sistema. Passamos por esta situação quando desenvolvemos um sistema para um cliente que já possuía um outro sistema desenvolvido por terceiros. O problema na hora de importar os dados é que, como existem muitas maneiras de se modelar tabelas de um banco de dados, e como cada programador pensa de um jeito diferente, as tabelas do banco origem às vezes são bem diferentes do banco destino. Outra situação parecida é a que

programadores mais experientes com certeza já passaram: ter que atualizar seu antigo sistema que armazenava dados em uma base de dados ultrapassada (*Paradox* por exemplo), para um banco de dados mais moderno. Nesse caso, o programador, com mais experiência do que quando criou o antigo sistema, encontra soluções mais elaboradas para estruturar o seu novo banco. A idéia desse artigo, para desenvolvedores iniciantes que certamente passarão por essa situação, é mostrar como criar uma aplicação em *Delphi* que irá acessar os dois bancos, percorrer registro por registro e inserir no banco destino fazendo todo controle de condições para manter a integridade dos dados.

Um exemplo de estruturas de tabelas diferentes seria:

• *Banco 1:* Contém uma tabela para armazenar clientes, outra para fornecedores e outra para funcionários;

• *Banco 2:* Contém uma tabela chamada *Pessoa*, onde são armazenados os vários tipos de cadastros com dados em comum, como por exemplo cliente, fornecedor e funcionário (têm em comum: nome, endereço, telefone, CPF, etc.). Para cada novo registro na tabela *Pessoa*, deve-se informar o valor *verdadeiro* ou *falso* para os campos *Cliente*, *Fornecedor* e *Funcionario*. Essa estrutura evita que uma pessoa que seja cliente e funcionário ao mesmo tempo seja cadastrada mais de uma vez.

No decorrer do artigo será colocada em prática a criação de dois bancos com características do exemplo citado acima. Vamos nos referenciar a eles como sendo respectivamente o banco origem (contém os dados já cadastrados) e o banco destino (receberá a importação dos dados). Ambos os bancos serão criados em *Firebird*. Na verdade, poderíamos exemplificar com *SGBDs* (*Sistema Gerenciador de Banco de Dados*) diferentes, mas o foco do artigo está na estrutura diferente das tabelas. Usando apenas o *Firebird* tornamos a prática do artigo mais dinâmica. Além do mais, todo o controle de importação será feito na aplicação, ou seja, não depende do *SGBD*.

Após a criação dos bancos, o próximo passo será a aplicação. Nesta, iremos primeiramente adicionar um formulário simples para inserir alguns cadastros no banco origem. Em seguida, outro formulário irá controlar todo o processo de importação dos dados, analisando as condições e inserindo os dados através de *DataSet´s*.

Criando a origem

Criaremos primeiramente um banco de dados simples para usar de exemplo para a origem dos dados que queremos importar. Este terá apenas as tabelas *Cliente*, *Dependente*, *Fornecedor*

Listagem 1. Script de criação do banco de dados origem

```
CREATE DATABASE 'C:\SISTEMAS\BANCOS\ORIGEM.FDB'
CREATE TABLE CLIENTE (
    CLI_CODIGO INTEGER NOT NULL,
    CLI_NOME VARCHAR(70) NOT NULL,
    CLI_PESSOA_FJ CHAR(1) NOT NULL,
    CLI_CPF_CNPJ VARCHAR(20) NOT NULL,
    CLI_ENDERECO1 VARCHAR(70),
    CLI_NUM_END1 VARCHAR(10),
    CLI_BAIRRO1 VARCHAR(40),
    CLI_CEP1 CHAR(10),
    CLI_CIDADE1 VARCHAR(40),
    CLI_UF1 CHAR(2),
    CLI_ENDERECO2 VARCHAR(70),
    CLI_NUM_END2 VARCHAR(10),
    CLI_BAIRRO2 VARCHAR(40),
    CLI_CEP2 CHAR(10),
    CLI_CIDADE2 VARCHAR(40),
    CLI_UF2 CHAR(2));
ALTER TABLE CLIENTE ADD CONSTRAINT PK_CLIENTE PRIMARY KEY (CLI_
CODIGO);
CREATE GENERATOR GEN_CLIENTE_ID;
CREATE TRIGGER T_CLIENTE_BI FOR CLIENTE ACTIVE BEFORE INSERT
POSITION 0
AS
BEGIN
  IF (NEW.CLI_CODIGO IS NULL) THEN
    NEW.CLI_CODIGO = GEN_ID(GEN_CLIENTE_ID,1);
END
;
CREATE TABLE DEPENDENTE (
    DEP_CODIGO     INTEGER NOT NULL,
    CLI_CODIGO     INTEGER NOT NULL,
    DEP_NOME       VARCHAR(70) NOT NULL,
    DEP_DATA_NASCIMENTO DATE
);
ALTER TABLE DEPENDENTE ADD CONSTRAINT PK_DEPENDENTE PRIMARY KEY
(DEP_CODIGO)
  USING INDEX PK_DEPENDENTE;
ALTER TABLE DEPENDENTE ADD CONSTRAINT FK_DEP_CLIENTE FOREIGN KEY
(CLI_CODIGO)
  REFERENCES CLIENTE(CLI_CODIGO)USING INDEX FK_DEP_CLIENTE;
CREATE GENERATOR GEN_DEPENDENTE_ID;
CREATE TRIGGER T_DEPENDENTE_BI FOR DEPENDENTE ACTIVE BEFORE INSERT
POSITION 0
AS
BEGIN
  IF (NEW.DEP_CODIGO IS NULL) THEN
    NEW.DEP_CODIGO = GEN_ID(GEN_DEPENDENTE_ID,1);
END
;
CREATE TABLE FORNECEDOR (
    FOR_CODIGO INTEGER NOT NULL,
    FOR_NOME VARCHAR(70) NOT NULL,
    FOR_PESSOA_FJ CHAR(1) NOT NULL,
    FOR_CPF VARCHAR(20) NOT NULL,
    FOR_ENDERECO VARCHAR(70),
    FOR_NUM_END VARCHAR(10),
    FOR_BAIRRO VARCHAR(40),
    FOR_CEP CHAR(10),
    FOR_CIDADE VARCHAR(40),
    FOR_UF CHAR(2)
);
ALTER TABLE FORNECEDOR ADD CONSTRAINT PK_FORNECEDOR PRIMARY KEY
(FOR_CODIGO);
CREATE GENERATOR GEN_FORNECEDOR_ID;
CREATE TRIGGER T_FORNECEDOR_BI FOR FORNECEDOR ACTIVE BEFORE INSERT
POSITION 0
AS
BEGIN
  IF (NEW.FOR_CODIGO IS NULL) THEN
    NEW.FOR_CODIGO = GEN_ID(GEN_FORNECEDOR_ID,1);
END
;
CREATE TABLE FUNCIONARIO (
    FUN_CODIGO INTEGER NOT NULL,
    FUN_NOME VARCHAR(70) NOT NULL,
    FUN_CPF VARCHAR(11) NOT NULL,
    FUN_ENDERECO VARCHAR(70),
    FUN_NUM_END VARCHAR(10),
    FUN_BAIRRO VARCHAR(40),
    FUN_CEP CHAR(10),
    FUN_CIDADE VARCHAR(40),
    FUN_UF CHAR(2),
    FUN_CARGO VARCHAR(40),
    FUN_SALARIO NUMERIC(12,2)
);
ALTER TABLE FUNCIONARIO ADD CONSTRAINT PK_FUNCIONRIO PRIMARY KEY
(FUN_CODIGO);
CREATE GENERATOR GEN_FUNCIONARIO_ID;
CREATE TRIGGER T_FUNCIONARIO_BI FOR FUNCIONARIO ACTIVE BEFORE
INSERT POSITION 0
AS
BEGIN
  IF (NEW.FUN_CODIGO IS NULL) THEN
    NEW.FUN_CODIGO = GEN_ID(GEN_FUNCIONARIO_ID,1);
END
;
```

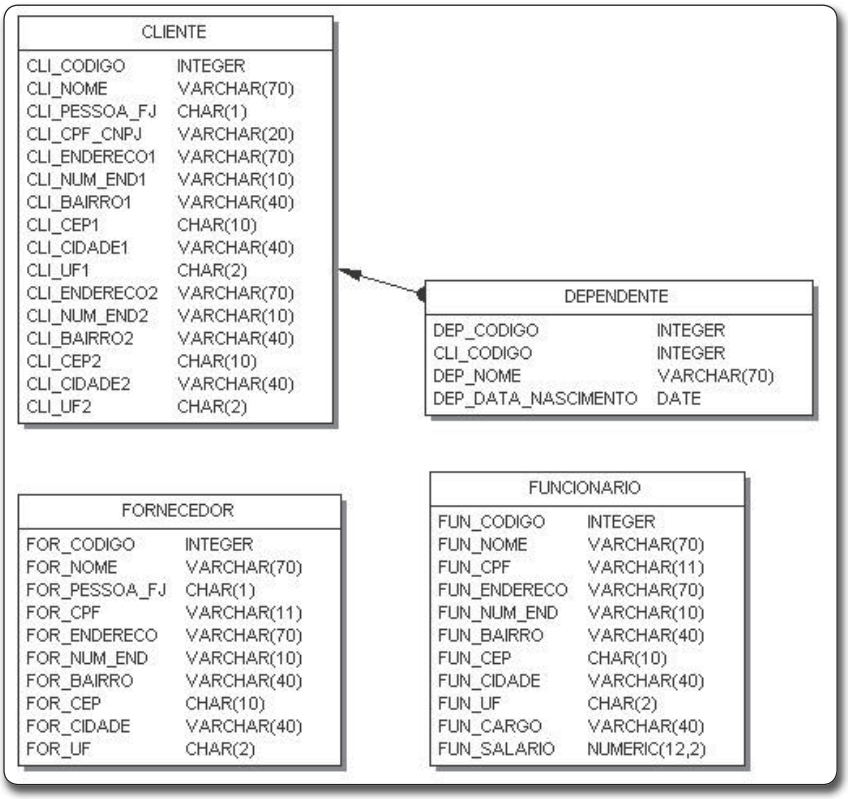


Figura 1. Diagrama com as tabelas do banco de dados origem

e *Funcionario*. Sugiro que o banco seja criado em *Firebird 2.0* e com a ferramenta *IBExpert* para execução dos *scripts*, mas como já foi dito, o *SGDB* fica a critério de cada um. A **Listagem 1** mostra o *script* para criação do banco de dados *origem*.

A **Listagem 2** traz um *script* opcional onde, para cada tabela, é criado um *genaretor* (*Gen_Cliente_Id*, *Gen_Dependente_Id*, etc.) e um *trigger* (*T_Cliente_Bi*, *T_Dependente_Bi*, etc.). O objetivo é gerar automaticamente o código da chave primária de cada tabela. Os *triggers* sempre serão disparados antes da confirmação de uma inserção (*Active Before Insert*) e irão atribuir a chave primária para a tabela respectiva, retornando o valor do *generator*.

Listagem 2. Script opcional para criação de triggers no banco de dados origem

```
CREATE GENERATOR GEN_CLIENTE_ID;
CREATE GENERATOR GEN_DEPENDENTE_ID;
CREATE GENERATOR GEN_FORNECEDOR_ID;
CREATE GENERATOR GEN_FUNCIONARIO_ID;
CREATE TRIGGER T_CLIENTE_BI FOR CLIENTE
ACTIVE BEFORE INSERT POSI TION 0
AS
BEGIN
  IF (NEW.CLI_CODIGO IS NULL) THEN
    NEW.CLI_CODIGO = GEN_ID(GEN_CLIENTE_ID,1);
END;
CREATE TRIGGER T_DEPENDENTE_BI FOR DEPENDENTE
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  IF (NEW.DEP_CODIGO IS NULL) THEN
    NEW.DEP_CODIGO = GEN_ID(GEN_DEPENDENTE_ID,1);
END;
CREATE TRIGGER T_FORNECEDOR_BI FOR FORNECEDOR
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  IF (NEW.FOR_CODIGO IS NULL) THEN
    NEW.FOR_CODIGO = GEN_ID(GEN_FORNECEDOR_ID,1);
END;
CREATE TRIGGER T_FUNCIONARIO_BI FOR FUNCIONARIO
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  IF (NEW.FUN_CODIGO IS NULL) THEN
    NEW.FUN_CODIGO = GEN_ID(GEN_FUNCIONARIO_ID,1);
END;
```

A tabela *Cliente* é bem simples e contém poucas informações, porém o suficiente para a compreensão do artigo. Note que foram criados dois campos para cada identificação complementar de endereço. Isso porque simulamos um sistema em que é necessário gravar os dados de dois tipos diferentes de endereço: endereço sede e endereço de entrega.

A tabela *Dependente* serve para armazenar os dependentes de um cliente, geralmente filhos e esposa. O campo *Cli_Codigo* na tabela *Dependente* é a chave estrangeira para manter o relacionamento com a tabela *Cliente* como pode ser visto na **Figura 1**.

Repare que muitas informações que são registradas para *Funcionario* e *Fornecedor* são também gravadas na tabela *Cliente*. Para essa estrutura, se um funcionário for também cliente, ele terá que ser cadastrado duas vezes (uma vez em cada tabela). No banco de dados *Destino* não haverá esse problema.

Criando o destino

O banco de dados *Destino* possui uma estrutura bem diferenciada do primeiro banco. Analisemos essas diferenças agora para podermos manter a integridade dos dados na hora de migrá-los. A **Listagem 3** mostra o *script* para a criação do novo banco e a **Figura 2** mostra um diagrama simples para uma visão geral do relacionamento entra as tabelas.

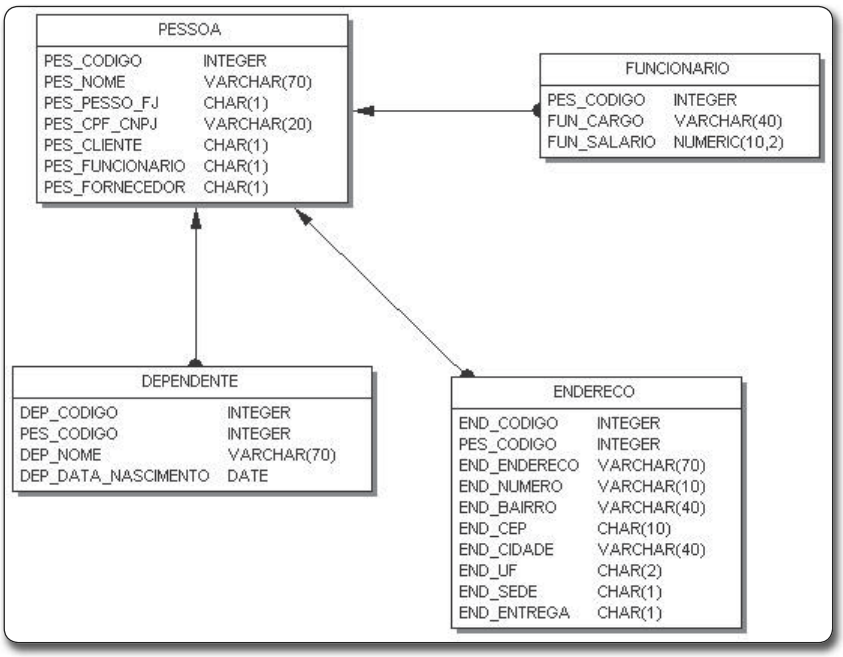


Figura 2. Diagrama com as tabelas do banco de dados destino

Listagem 3. Script de criação do banco de dados DESTINO

```
CREATE DATABASE 'C:\SISTEMAS\BANCOS\DESTINO.FDB'
CREATE TABLE PESSOA (
  PES_CODIGO INTEGER NOT NULL,
  PES_NOME VARCHAR(70) NOT NULL,
  PES_PESSO_FJ CHAR(1) NOT NULL,
  PES_CPF_CNPJ VARCHAR(20) NOT NULL,
  PES_CLIENTE CHAR(1) NOT NULL,
  PES_FUNCIONARIO CHAR(1) NOT NULL,
  PES_FORNECEDOR CHAR(1) NOT NULL);
ALTER TABLE PESSOA ADD CONSTRAINT PK_PESSOA PRIMARY KEY (PES_
CODIGO);
CREATE TABLE ENDERECO (
  END_CODIGO INTEGER NOT NULL,
  PES_CODIGO INTEGER NOT NULL,
  END_ENDERECO VARCHAR(70) NOT NULL,
  END_NUMERO VARCHAR(10) NOT NULL,
  END_BAIRRO VARCHAR(40) NOT NULL,
  END_CEP CHAR(10) NOT NULL,
  END_CIDADE VARCHAR(40) NOT NULL,
  END_UF CHAR(2) NOT NULL,
  END_SEDE CHAR(1) NOT NULL,
  END_ENTREGA CHAR(1) NOT NULL);
CREATE TABLE FUNCIONARIO (
    PES_CODIGO INTEGER NOT NULL,
    FUN_CARGO VARCHAR(40) NOT NULL,
    FUN_SALARIO NUMERIC(10,2) NOT NULL
);
ALTER TABLE FUNCIONARIO ADD CONSTRAINT PK_FUNCIONARIO PRIMARY KEY
(PES_CODIGO);
ALTER TABLE FUNCIONARIO ADD CONSTRAINT FK_FUN_PESSOA FOREIGN KEY
(PES_CODIGO)
REFERENCES PESSOA(PES_CODIGO)ON DELETE CASCADE ON UPDATE
CASCADEUSING INDEX
  FK_FUN_PESSOA;
ALTER TABLE ENDERECO ADD CONSTRAINT PK_ENDERECO PRIMARY KEY (END_
CODIGO);
ALTER TABLE ENDERECO ADD CONSTRAINT FK_END_PESSOA FOREIGN KEY
(PES_CODIGO)REFERENCES
  PESSOA(PES_CODIGO)ON DELETE CASCADE ON UPDATE CASCADE USING
INDEX FK_END_PESSOA;
CREATE TABLE DEPENDENTE (
  DEP_CODIGO     INTEGER NOT NULL,
  PES_CODIGO     INTEGER NOT NULL,
  DEP_NOME       VARCHAR(70) NOT NULL,
  DEP_DATA_NASCIMENTO DATE
);
ALTER TABLE DEPENDENTE ADD CONSTRAINT PK_DEPENDENTE PRIMARY KEY
(DEP_CODIGO) USING
  INDEX PK_DEPENDENTE;
ALTER TABLE DEPENDENTE ADD CONSTRAINT FK_DEP_PESSOA FOREIGN KEY
(PES_CODIGO)
  REFERENCES PESSOA(PES_CODIGO)USING INDEX FK_DEP_PESSOA;
CREATE GENERATOR GEN_PESSOA_ID;
CREATE GENERATOR GEN_ENDERECO_ID;
CREATE GENERATOR GEN_DEPENDENTE_ID;
```

Não foram criadas *triggers* para o banco *Destino*, somente os *genarators*. Isso será explicado mais à frente no artigo durante a demonstração da aplicação para migrar os dados.

Primeiramente vamos considerar que a tabela que armazena os dados dos clientes se chama agora *Pessoa*. Existe a tabela *Endereco* que tem uma chave estrangeira com o campo *Pes_Codigo* para manter o relacionamento com a tabela *Pessoa*. Assim pode-se cadastrar um ou mais endereços para a *Pessoa* informando se o endereço é sede e/ou endereço de entrega (marcando os campos *End_Sede* e *End_Entrega*).

Não foi criada a tabela *Fornecedor* como no banco *Origem*. Isso porque a tabela *Pessoa* irá armazenar os dados dos fornecedores, assim como de clientes e também de funcionários. Lembremos que no banco *Origem*, a tabela *Fornecedor* armazena os mesmos dados que a tabela *Cliente*. Agora, na tabela *Pessoa*, os campos *Pes_Cliente*, *Pes_Funcionario* e *Pes_Fornecedor* respectivamente servem para informar se o registro é um cliente e/ou funcionário e/ou fornecedor. Ambos os campos são do tipo *char(1)*, ou seja, permitem apenas um caractere alfanumérico, que serão configurados com os valores *V* (verdadeiro) e *F*(falso) – controle geralmente feito por um *CheckBox* na aplicação. Sendo assim, se a empresa tem um fornecedor que é também um cliente, este será cadastrado uma só vez marcando *V* para os campos *Pes_Cliente* e também para o campo *Pes_Fornecedor*.

A tabela *Dependente* se mantém como no antigo banco de dados. Note que foi necessária a criação da tabela *Funcionario*. Isso

porque, diferente de clientes e fornecedores, existem particularidades para o registro de funcionários, que é a informação do cargo e a do salário. A tabela *Funcionario* funciona como uma tabela auxiliar, de relacionamento 1-1 (um para um) com a tabela *Pessoa*. Armazena dados somente quando um registro na tabela *Pessoa* tiver o campo *Pes_Funcionario* contendo o valor *V* (*verdadeiro*).

Criando uma aplicação para migrar os dados

A aplicação a ser criada será feita sem nenhuma preocupação com a estética visual, pois sua função é de uso exclusivo do programador e não será apresentada a um usuário de sistema. Terá apenas três formulários: o menu principal para fazer a chamada dos demais; outro para cadastrar alguns dados de exemplo para o banco origem; e o formulário responsável pela migração dos dados que é o grande foco do projeto. O exemplo será construído em Delphi 7.0 usando o componente dbExpress para fazer a conexão com o banco de dados, mas poderá ser usada qualquer outra versão que apresente compatibilidade de conexão com o *SGBD* usado.

Abra o Delphi e crie uma nova aplicação através do menu *File>New>Application.* Mude a propriedade *Name* do formulário para *fmPrincipal* e salve a *unit* como *uPrincipal.pas*. Agora salve o projeto como *Migracao.dptr*. Insira no *fmPrincipal* um *Button* e mude nome para *btnCadastros* e o *caption* para *Cadastros*. Agora outro com o nome de *btnMigracao* e o *Caption Migração* como mostra a **Figura 3**.

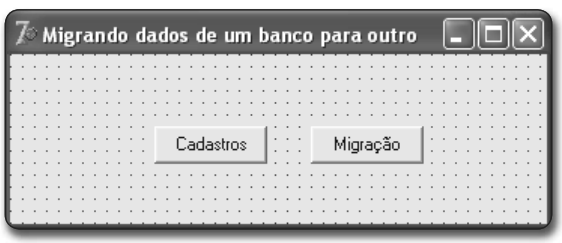


Figura 3. Fomulário principal da aplicação

Agora adicione um *DataModule* (*File>New>Data Module*), nomeie-o para *dmO*, e salve a *unit* como *uDmO.pas*. Adicione um *SQLConnection* da paleta de componentes *dbExpress* e nomeie-o para *SQLConnectionOrigem*. Agora insira quatro componentes *SQLDataSet* (da paleta *dbExpress*). Mude para a paleta *Data Access*, e insira três *DataSetProviders*, quatro *ClientDataSets* e um *DataSource*. Posicione e nomeie-os como mostra a **Figura 4**.

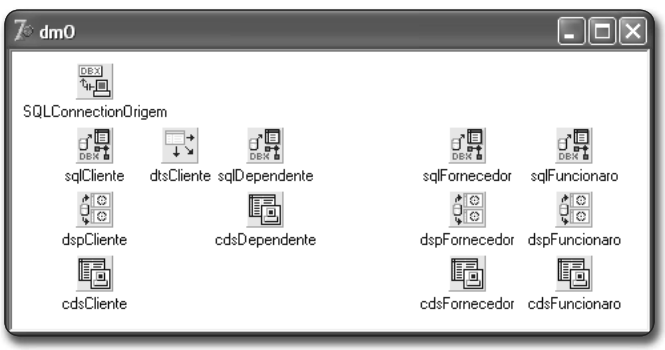


Figura 4. Data Module de conexão com o banco de dados origem

Para configurar a conexão com o banco de dados origem dê um duplo-clique no *SQLConnectionOrigem*, adicione uma nova conexão através do botão *Add Connection* (representado por um sinal de soma); para a opção *Driver Name* selecione *Interbase*, e para *Connection Name* digite *origem*. Como mostra a **Figura 5**, resta agora digitar o caminho do banco na propriedade *Data Base* da grade *Connection Settings*, *User_Name* para *sysdba* e *Password* para *masterkey* (usuário e senha que foram definidos no *script* de criação do banco de dados origem), e por último mudar o *SQLDialect* para 3 e confirmar no botão *Ok*. Mude a propriedade *LoginPrompt* do *SQLConnectionOrigem* para *False*, assim não será necessário digitar a senha sempre que houver conexão com o banco.

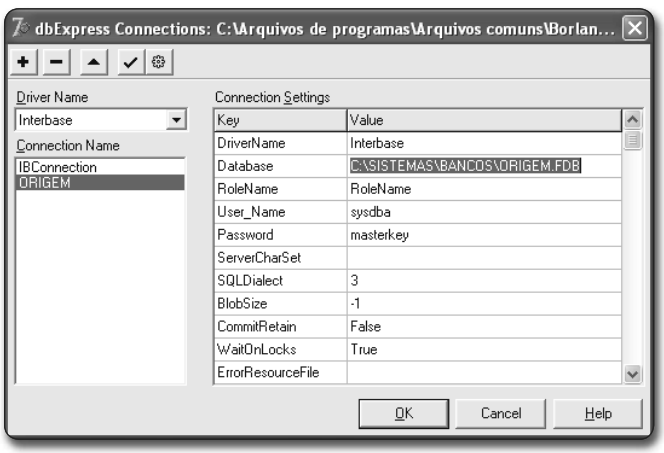


Figura 5. Configuração da conexão com o banco de dados ORIGEM

Abaixo seguem as configurações para ligar os componentes:

- Mude a propriedade *SQLConnection* de todos os *SQLDataSets* para *SQLConnectionOrigem*;
- Ligue a propriedade *DataSet* do *dspCliente* em *sqlCliente*, a propriedade *ProviderName* do *cdsCliente* em *dspCliente*;
- Ligue a propriedade *DataSet* do *dspFornecedor* em *sqlFornecedor* e a propriedade *ProviderName* do *cdsFornecedor* em *dspFornecedor*;
- Ligue a propriedade *DataSet* do *dspFuncionario* em *sqlFuncionario* e a propriedade *ProviderName* do *cdsFuncionario* em *dspFuncionario;*
- Para poder criar um vínculo de dependentes com clientes, os componentes referentes aos dependentes serão configurados mais adiante.

Insira as *Selects* da **Listagem 4** na propriedade *CommandText* dos seus respectivos *SQLDataSets* (exemplo: a SELECT com comentário */* sqlCliente */* deve ser inserida no *sqlCliente*). As *Selects* são bem simples e trazem todos os dados de uma só vez, o que não é aconselhável. Mas a aplicação que está sendo criada não vai criar tráfego em rede e vai ser usada uma única vez, ou seja, depois de fazer a importação corretamente a aplicação não será mais necessária.

Listagem 4. SELECTs para cadastro do banco de dados ORIGEM

```
/* sqlCliente */
SELECT
  CLIENTE.CLI_CODIGO, CLIENTE.CLI_NOME, CLIENTE.CLI_PESSOA_FJ,
   CLIENTE.CLI_CPF_CNPJ,
  CLIENTE.CLI_ENDERECO1, CLIENTE.CLI_NUM_END1, CLIENTE.CLI_
   BAIRRO1, CLIENTE.CLI_CEP1,
  CLIENTE.CLI_CIDADE1, CLIENTE.CLI_UF1, CLIENTE.CLI_ENDERECO2,
   CLIENTE.CLI_NUM_END2,
  CLIENTE.CLI_BAIRRO2, CLIENTE.CLI_CEP2, CLIENTE.CLI_CIDADE2,
   CLIENTE.CLI_UF2
FROM
  CLIENTE

/* sqlDependente */
SELECT
  DEPENDENTE.DEP_CODIGO, DEPENDENTE.CLI_CODIGO, DEPENDENTE.DEP_NOME,
  DEPENDENTE.DEP_DATA_NASCIMENTO
FROM
  DEPENDENTE
WHERE
  DEPENDENTE.CLI_CODIGO = :CLI_CODIGO

/* sqlFornecedor */
SELECT
  FORNECEDOR.FOR_CODIGO, FORNECEDOR.FOR_NOME, FORNECEDOR.
   FOR_PESSOA_FJ,
  FORNECEDOR.FOR_CPF, FORNECEDOR.FOR_ENDERECO, FORNECEDOR.
   FOR_NUM_END,
  FORNECEDOR.FOR_BAIRRO, FORNECEDOR.FOR_CEP, FORNECEDOR.
   FOR_CIDADE, FORNECEDOR.FOR_UF
FROM
  FORNECEDOR

/* sqlFuncionário */
SELECT
  FUNCIONARIO.FUN_CODIGO, FUNCIONARIO.FUN_NOME, FUNCIONARIO.
   FUN_CPF,
  FUNCIONARIO.FUN_ENDERECO, FUNCIONARIO.FUN_NUM_END,
   FUNCIONARIO.FUN_BAIRRO,
  FUNCIONARIO.FUN_CEP, FUNCIONARIO.FUN_CIDADE, FUNCIONARIO.
   FUN_UF,
  FUNCIONARIO.FUN_CARGO, FUNCIONARIO.FUN_SALARIO
FROM
  FUNCIONARIO
```

Já com todas as *Selects* em seus respectivos *SQLDataSets*, clique com o botão direito sobre o *sqlCliente*, depois clique em *Fields Editor*. Dentro do *Fields Editor*, clique com o botão direito e depois clique na opção *Add all fields*.

Para quem optou por gerar automaticamente a chave das tabelas através dos *triggers*, clique sobre o *field Cli_Codigo* na lista de *fields* e marque a propriedade *Required* para *False* e a propriedade *ReadOnly* pra *True*. Isso é necessário porque vamos deixar o campo com o valor nulo para que seja gerado no banco de dados, assim a aplicação não acusará a falta de valor para um campo requerido. Caso tenha ignorado a opção da criação dos *triggers*, o campo *Cli_Codigo*, e as demais chaves primárias deverão ser informado a cada novo registro.

Repita a operação para os outros *SQLDataSets*. Agora insira os *fields* nos *ClientDataSets* da mesma forma que foi inserido no *SQLDataSets*, menos para *cdsDependente*.

Abra a propriedade *Params* do *sqlDependente*, onde aparecerá o parâmetro *Cli_Codigo*. Mude sua propriedade *DataType* para *ftInteger*. Agora adicione todos os *fields* para o *sqlDependente*. Ligue a propriedade do *DataSource* do *sqlDependente* em *sqlCliente*. Feita essa última configuração, será necessário ativar o *sqlCliente* para poder ativar o *sqlDependente*. Pois ao ligarmos o *sqlDependente* no *DataSource sqlCliente* criamos um vínculo entre os dois. Volte agora no *Fields Editor* do *cdsCliente* (botão direito, *Fields Editor*) e acione a opção *Add all fields* com o botão direito. Aparecerá agora um novo *field* chamado *sqlDependente*. Agora ligue a propriedade *DataSetField* do *cdsDependente* em *cdsClientesqlDependente*. Em seguida adicione todos os *fields* para o *cdsDependente*. Esse tipo de relacionamento criado entre clientes e dependentes chama-se *Mestre-Detalhe*.

Caso esteja utilizando os *triggers*, clique sobre o *field Dep_Codigo* na lista de *fields* e marque a propriedade *Required* para *False* e a propriedade *ReadOnly* pra *True*. Para poupar código, deixe a propriedade *Active* de todos os *ClientDataSets* como *True*. Adicione um novo formulário (*File>New>Form*) e nomeie-o para *frmCadastros* e salve a *unit* como *uCadastros*. Volte para o *frmPrincipal* e adicione o *frmCadastros* (*Alt+F11*). Dê um duplo-clique no *btnCadastros* e insira o seguinte código para fazer a chama do *frmCadastros*:

```
  frmCadastros := TfrmCadastros.Create(Self);
try
  frmCadastros.ShowModal;
finally
  FreeAndNil(frmCadastros);
end;
```

O formulário de cadastro, como já foi dito, não é o foco do projeto e será configurado com os recursos mais simples possíveis que se permitam gerar cadastros para o exemplo do artigo. Com certeza não deve ser seguido como exemplo para implantação em um sistema comercial. Insira primeiramente no *frmCadastros* um componente *PageControl* da paleta *Win32*. Mude sua propriedade *Align* para *alClient*, em seguida adicione três páginas ao *PageControl* clicando com o botão direto e selecionando a opção *NewPage*. Mude a propriedade *Caption* de cada página respectivamente para *Fornecedor, Funcionário* e *Cliente*. Vamos configurar a aba do cadastro de funcionário antes da aba de clientes por ser a mais fácil. Insira nela um componente *DBNavigator* e um *DBGrig* da paleta *Data Controls* e um *Button* com o *caption Aplicar Alterações no Banco*, como mostra a **Figura 6**.

Agora insira quatro *DataSources* e nemeie-os como *dtsCliente*, *dtsDependente*, *dtsFornecedor* e *dtsFuncionario*.

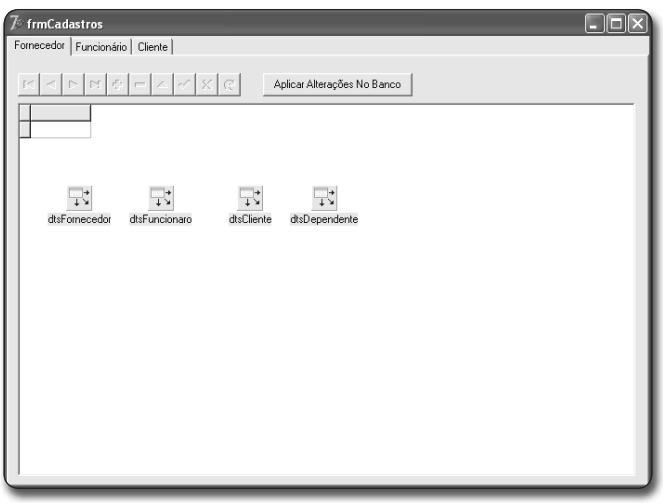


Figura 6. Fomulário de cadastros

Estando o *frmCadastros* em primeiro plano, aperte *Alt+F11* e selecione a *unit dmO*. Agora podemos ligar a propriedade *DataSet* do *dtsFornecedor* em *dmO.cdsFornecedor* e o restante dos *DataSources* do *frmCadastros* nos seus respectivos *ClientDataSets*.

Ligue as propriedades *DataSource* do *DBGrid* e do *DBNavigator* em *dtsFornecedor*. Agora dê um duplo-clique no *Button* e insira o seguinte código:

```
if (dm0.cdsFornecedor.ApplyUpdates(0) = 0) then
begin
  dm0.cdsFornecedor.Close;
  dm0.cdsFornecedor.Open;
  ShowMessage('Dados Confirmados!');
end;
```

Este código envia as alterações feitas previamente no *ClientDataSet* para o banco de dados validar e, se não houver erros, *commitar*.

Poderemos agora executar alguns cadastros para usar no exemplo, através do *DBNavigator* para inserir, editar e excluir, e da *DBGrid* para escrever os dados. Não se esqueça de clicar no botão que confirma os dados no banco, senão toda alteração será descartada. Agora é só repetir os passos para configurar a aba do cadastro de Funcionários e de Clientes.

Confira nas **Figuras 7, 8 e 9** como fica cada aba de cadastro. É só ligar os componentes aos seus respectivos controles como foi feito para o cadastro de fornecedores.

Há uma regra para o cadastro de dependentes, que é confirmar os dados do cliente (no botão Aplicar Alterações no Banco) antes de inserir os seus dependentes. Fazendo assim, quando houver uma inserção de dependente o registro já virá com o código do cliente - controle feito pelo relacionamento Mestre Detalhe.

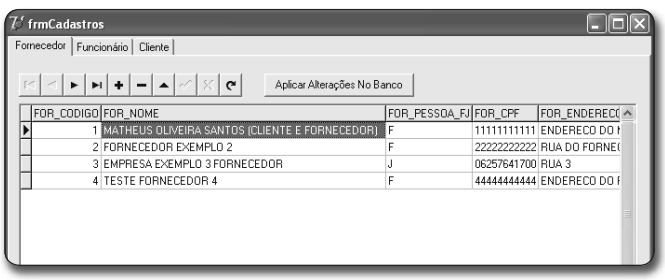


Figura 7. Fomulário de cadastros – Cadastro de Fornecedores

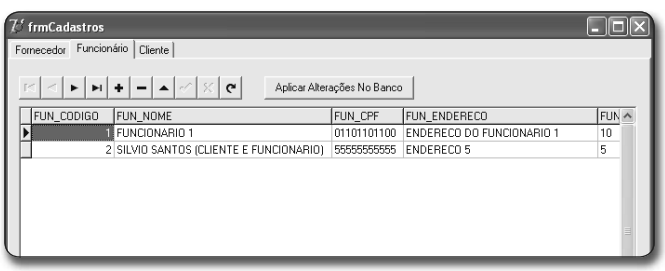


Figura 8. Fomulário de cadastros – Cadastro de Funcionários

Cadastre alguns dados simulando que um funcionário seja também cliente, ou um fornecedor. Nesse caso devemos repetir as informações nas tabelas, principalmente o *CPF*, que será importante para a migração. Para o caso do endereço do cliente, que é composto por endereço sede (*Cli_Endereco1*) e endereço de entrega (*Cli_Endereco2*), se um cliente possuir um único endereço, deixe o segundo em branco ou repita as informações.

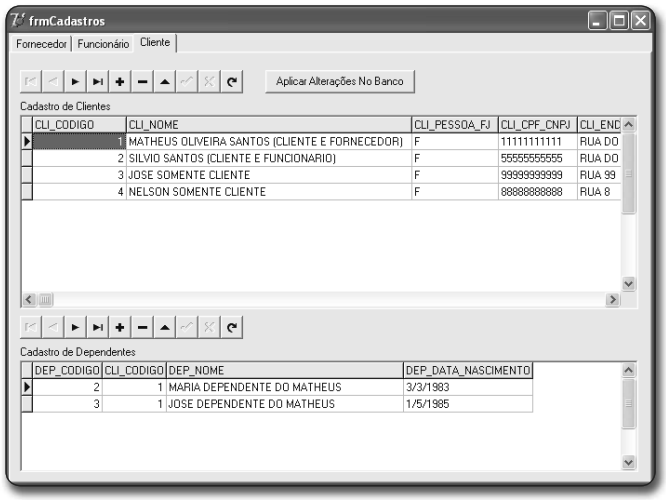


Figura 9. Fomulário de cadastros – Cadastro de Clientes

Agora vamos adicionar outro *Data Module* no projeto e nomeá-lo como *dmD* e salvar a *unit* como *uDmD*. Adicione um *SQLConnection* e nomeie como *SQLConnectionDestino*. Para criar uma conexão com o banco de dados destino, siga os mesmos passos necessários para criar a conexão com o banco origem, mudando apenas o diretório do banco.

Finalmente, o formulário que irá migrar os dados de um banco para outro. Adicione outro formulário no projeto e nomeie-o para *frmMigracao* e salve a *unit* como *uMigracao*. Volte para o *frmPrincipal* e adicione o *frmMigracao* (*Alt+F11*). Dê um duplo-clique no *btnMigracao* e insira o seguinte código para fazer a chama do *frmMigracao*:

```
frmMigracao := TfrmMigracao.Create(Self);
try
  frmMigracao.ShowModal;
finally
  FreeAndNil(frmMigracao);
end;
```

Insira no *frmMigracao* um *PageControl* e mude sua propriedade *Align* para *alClient*, em seguida adicione três páginas ao *PageControl* clicando com o botão direto e selecionando a opção *NewPage*. Mude a propriedade *Caption* de cada página respectivamente para *Cliente*, *Fornecedor* e *Funcionário*.

Primeiramente, antes de pensar em migrar, vamos inserir os componentes que possibilitem a visualização dos dados existentes em ambos os bancos. Insira quatro *DBGrids* na aba de clientes, quatro *Labels*, um *Button*, quatro *DataSources*, dimensione e mude o *caption* dos componentes como mostra a **Figura 10**.

Mude para a aba de Fornecedores, insira e nomeie os componentes como mostra a **Figura 11**. A **Figura 12** mostra como configurar a aba de Funcionários. Não achei necessário exibir os endereços de fornecedores e funcionários, mas os mesmos serão migrados normalmente.

Para fazermos a importarção dos clientes, vamos listar todos os clientes do banco origem, que aparecerão no *DBGrid* Clientes origem, percorrer todos os registros e, a cada registro, vamos inserir os dados na tabela pessoa do banco destino. Os clientes importados aparecerão no *DBGrid* clientes destino.

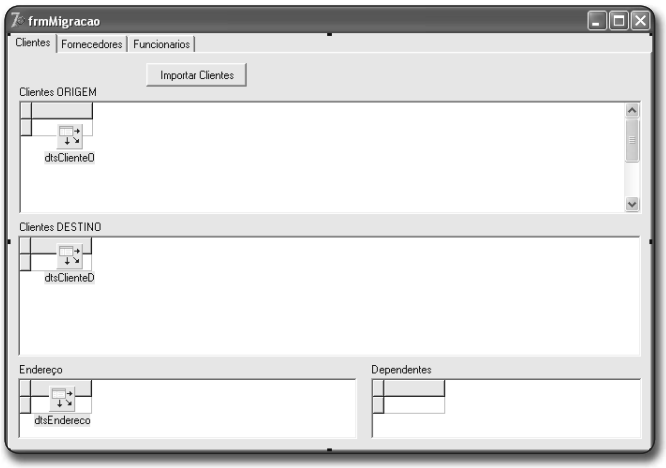


Figura 10. Formulário de importação dos dados – Aba Clientes

Figura 12. Formulário de importação dos dados – Aba Funcionários

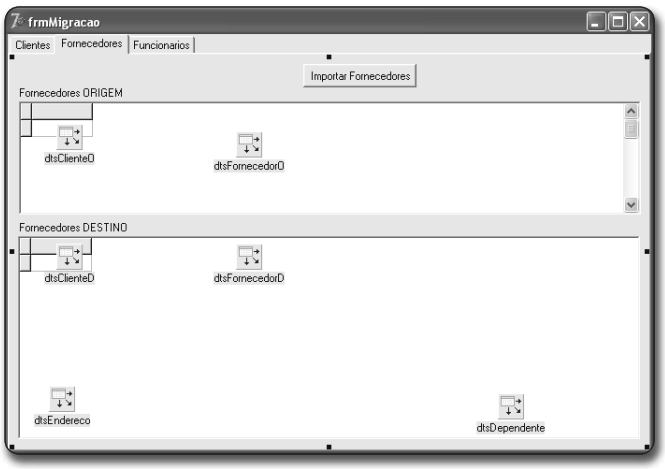


Figura 11. Formulário de importação dos dados – Aba Fornecedores

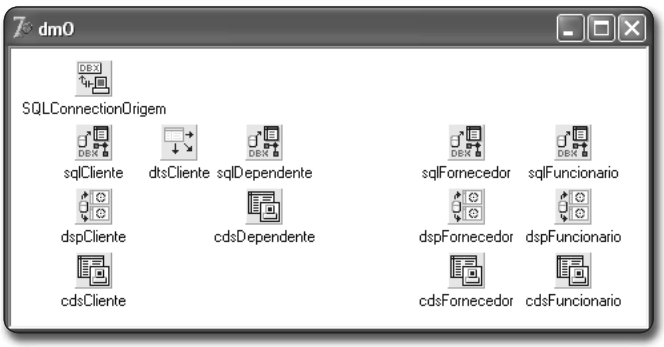


Figura 13. Data Module de conexão com o banco de dados destino

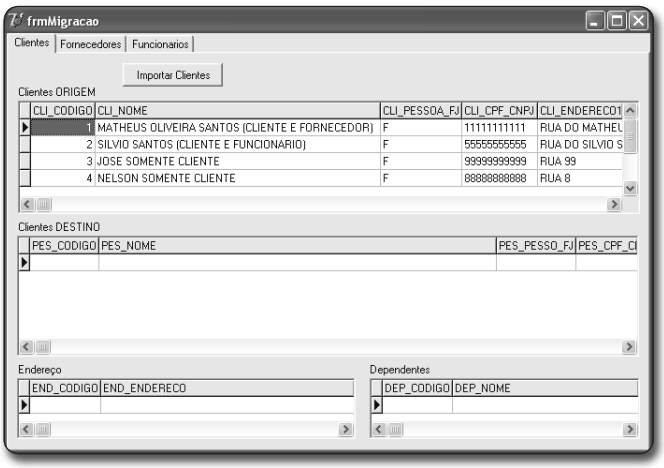


Figura 14. Formulário de Migração

Vamos agora inserir os componentes no *dmD* que, depois dos dados migrados, retornarão a listagem de todos os dados dos clientes, para serem mostrados nos *DBGrids* que inserimos no *frmMigracao*. Insira cinco *SQLDataSets*, três *DataSetProviders*, cinco *ClientDataSets* e um *DataSource*. Posicione e nomei-os como mosta a **Figura 13**.

Abaixo seguem as configurações para ligar os componentes:

• Mude a propriedade *SQLConnection* de todos os *SQLDataSets* para *SQLConnectionDestino*;

• Insira as SELECTs da **Listagem 5** na propriedade *CommandText* dos seus respectivos *SQLDataSets* (exemplo: a SELECT com comentário /* *sqlCliente* */ em *sqlCliente*);

• Ligue a propriedade *DataSet* do *dspCliente* em *sqlCliente*, a propriedade *ProviderName* do *cdsCliente* em *dspCliente*;

• Ligue a propriedade *DataSet* do *dspFornecedor* em *sqlFornecedor* e a propriedade *ProviderName* do *cdsFornecedor* em *dspFornecedor*;

• Ligue a propriedade *DataSet* do *dspFuncionario* em *sqlFuncionario* e a propriedade *ProviderName* do *cdsFuncionario* em *dspFuncionario;*

• Adicione todos os *fields* no *Fields Editor* do *sqlCliente* (botão direito em *sqlCliente*, clique em *Fields Editor*, botão direito em *Fields Editor*, clique em *Add all fields*).Adicione todos os *fields* no *Fields Editor* do *cdslClient*. Repita a operação para fornecedor e funcionário;

• Configure o parâmetro *Pes_Codigo* do *sqlEndereco* como *ftInteger* (entre na propriedade Params do *sqlEndereco*, selecione o parâmetro *Pes_Codigo* na lista de parâmetros e mude o *DataType* do mesmo para *ftInteger*). Repita a operação para o *sqlDependente*;

• Ligue a propriedade *DataSet* do *dtsCliente* em *sqlCliente*. Ligue a propriedade *DataSource* do *sqlEndereco* e também do *sqlDependente* em *dtsCliente*. Mude a propriedade *Active* do

sqlCliente para *True*. Adicione todos os *fields* para *sqlEndereco* e também para *sqlDependente*. Mude a propriedade *Active* do *sqlCliente* para *False*;
• Faça o processo de adicionar todos os *fields* para *cdsCliente* novamente. Deve aparecer dois novos *fields*: *sqlEnderco* e *sqlDependente*;
• Mude a propriedade *DataSetField* do *cdsEndereco* para *cdsClientesqlEnderco*. Mude a propriedade *DataSetField* do *cdsDependente* para *cdsClientesqlDependente*;
• Adicione todos os *fields* para *cdsEndereco* e *cdsDependente*;
• Desative todos os *SQLDataSets* e deixe ativo os *ClientDataSets* (propriedade *Active*).

Voltemos ao *frmMigracao*. Adicione a ele o *dmO* e o *dmD* (*Alt+F11*). Agora ligue a propriedade *DataSet* de cada *DataSource* em seu respectivo *ClientDataSet*:
• *dtsClienteO* em *dmO.cdsCliente*, *dtsClienteD* em *dmD.cdsCliente*, *dtsEndereco* em *dmD.cdsEndereco*, *dtsDependente* em *dmD.cdsDependente*.
• Como não demos nome aos *DBGrids*, vamos referenciá-los pelos *labels* que estão logo acima de cada um. Sendo assim, ligue a propriedade *DataSource* do *DBGrid* clientes destino em *dtsClienteO*, *DBGrid* Clientes destino em *dtsClienteD*, endereço em *dtsEndereco*, dependentes em *dtsDependente*.

Listagem 5. Lista de SELECTs para o Data Module Dmd

```
/* sqlCliente */
SELECT
  PESSOA.PES_CODIGO, PESSOA.PES_NOME, PESSOA.PES_PESSO_FJ,
   PESSOA.PES_CPF_CNPJ
FROM
  PESSOA
WHERE
  PESSOA.PES_CLIENTE = 'V'

/* sqlEndereco */
SELECT
  ENDERECO.END_CODIGO, ENDERECO.END_ENDERECO, ENDERECO.END_NUMERO,
  ENDERECO.END_BAIRRO, ENDERECO.END_CEP, ENDERECO.END_CIDADE,
  ENDERECO.END_UF,
  ENDERECO.END_SEDE, ENDERECO.END_ENTREGA
FROM
  ENDERECO
WHERE
  ENDERECO.PES_CODIGO = :PES_CODIGO

/* sqlDependente */
SELECT
  DEPENDENTE.DEP_CODIGO, DEPENDENTE.DEP_NOME, DEPENDENTE.DEP_
   DATA_NASCIMENTO
FROM
  DEPENDENTE
WHERE
  DEPENDENTE.PES_CODIGO = :PES_CODIGO

/* sqlFornecedor */
SELECT
  PESSOA.PES_CODIGO, PESSOA.PES_NOME, PESSOA.PES_PESSO_FJ,
   PESSOA.PES_CPF_CNPJ
FROM
  PESSOA
WHERE
  PESSOA.PES_FORNECEDOR = 'V'

/* sqlFuncionario */
SELECT
  PESSOA.PES_CODIGO, PESSOA.PES_NOME, PESSOA.PES_PESSO_FJ,
   PESSOA.PES_CPF_CNPJ
  FUNCIONARIO.FUN_CARGO, FUNCIONARIO.FUN_SALARIO
FROM
  PESSOA
LEFT JOIN FUNCIONARIO ON
  PESSOA.PES_CODIGO = FUNCIONARIO.PES_CODIGO
WHERE
  PESSOA.PES_FUNCIONARIO = 'V'
```

O que temos até agora no *frmMigracao* nos possibilita visualizar os dados dos clientes de cada banco, como pode ser visto na **Figura 14**. Claro que só temos os dados cadastros no banco origem, mas estamos deixando a visualização dos dados pronta para depois nos concentrarmos na migração.

Configuraremos agora a visualização dos fornecedores e funcionários:
• Ligue o *dtsFornecedorO* em *dmO.cdsFornecedor*, o *DBGrid* Fornecedores origem em *dtsFornecedorO*, ligue o *dtsFornecedorD* em *dmD.cdsFornecedor*, o *DBGrid* Fornecedores destino em *dtsFornecedorD*.
• Ligue o *dtsFuncionarioO* em *dmO.cdsFuncionario*, o *DBGrid* Funcionários origem em *dtsFuncionarioO*, ligue o *dtsFuncionarioD* em *dmD.cdsFuncionario*, o *DBGrid* funcionários destino em *dtsFuncionarioD*.

Enfim, vamos começar a implantar as soluções para migrar os dados. Volte para o *dmD* e insira sete *SQLDataSets*, ligue a propriedade *SQLConection* de todos em *SQLConnectionDestino*, posicione e nomeie-os como mostra a **Figura 15**. A **Listagem 6** mostra as *selects* a serem inseridas nos seus respectivos *SQLDataSets* (ex.: inserir a *Select* `/* sqlChavePessoa */` em *sqlChavePessoa*).

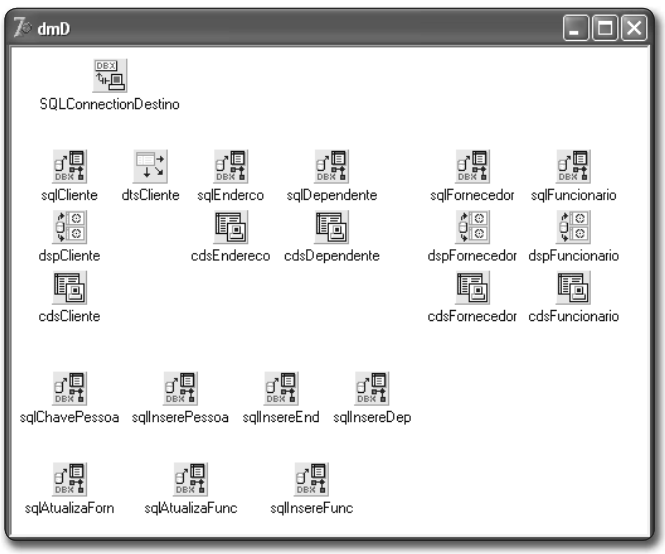


Figura 15. dmD - Data Module DESTINO

Configurando os novos componentes:
• Clique com o botão direito em *sqlChavePessoa*, clique em *Fields Editor*, clique em *Add all fields*. Agora temos o *field Pes_Codigo* que será usado sempre que precisarmos atribuir a chave primária a tabela PESSOA.
• Abra a propriedade *Params* do *sqlInserePessoa* e configure a propriedade *Data Type* do parâmetro *Pes_Codigo* como *ftInteger*, e o restante dos parâmetros como *ftString*.
• Abra a propriedade *Params* do *sqlInsereEnd* e configure a propriedade *Data Type* do parâmetro *Pes_Codigo* como *ftInteger*, e o restante dos parâmetros como *ftString*.
• Abra a propriedade *Params* do *sqlInsereDep* e configure a

propriedade *Data Type* do parâmetro *Pes_Codigo* como *ftInteger*, *Dep_Nome* como *ftString*, e *Dep_Data_Nascimento* como *ftDate*.
• Abra a propriedade *Params* do *sqlAtualizaForn* e configure a propriedade *Data Type* do parâmetro *Pes_Codigo* como *ftInteger*.
• Abra a propriedade *Params* do *sqlAtualizaFunc* e configure a propriedade *Data Type* do parâmetro *Pes_Codigo* como *ftInteger*.
• Abra a propriedade *Params* do *sqlInsereFunc* e configure a propriedade *Data Type* do parâmetro *Pes_Codigo* como *ftInteger*, *Fun_Cargo* como *ftString*, e *Fun_Salario* como *ftCurrency*.

Não será possível importar a chave primária, pois lembremos que, além da tabela cliente, existem os dados da tabela fornecedor e da tabela funcionario do banco origem que também deverão ser inseridos na tabela pessoa do banco destino. Nesse caso pode haver, por exemplo, um cliente com o código 2 e um funcionário com o código 2. Sendo assim, como a chave primária deve se única e exclusiva, iremos atribuir para o campo pes_codigo (chave primária da tabela pessoa) um novo código seqüencial para cada registro inserido na tabela.

Listagem 6. Lista de SELECTs para novos componentes do Data Module dmD

```
/* sqlChavePessoa */
SELECT GEN_ID(GEN_PESSOA_ID, 1) AS PES_CODIGO
FROM
  RDB$DATABASE

/* sqlInserePessoa */
INSERT INTO PESSOA (
  PESSOA.PES_CODIGO, PESSOA.PES_NOME, PESSOA.PES_PESSO_FJ,
  PESSOA.PES_CPF_CNPJ,
  PESSOA.PES_CLIENTE, PESSOA.PES_FUNCIONARIO, PESSOA.PES_FORNECEDOR
)
VALUES (
  :PES_CODIGO, :PES_NOME, :PES_PESSO_FJ, :PES_CPF_CNPJ, :PES_CLIENTE,
  :PES_FUNCIONARIO, :PES_FORNECEDOR
)

/* sqlInsereEnd */
INSERT INTO ENDERECO (
  ENDERECO.END_CODIGO, ENDERECO.PES_CODIGO, ENDERECO.
  END_ENDERECO,ENDERECO.END_NUMERO,
  ENDERECO.END_BAIRRO, ENDERECO.END_CEP,
  ENDERECO.END_CIDADE,ENDERECO.END_UF,
  ENDERECO.END_SEDE,ENDERECO.END_ENTREGA
)
VALUES (
  GEN_ID(GEN_ENDERECO_ID, 1), :PES_CODIGO, :END_ENDERECO,
    :END_NUMERO, :END_BAIRRO,
  :END_CEP, :END_CIDADE, :END_UF, :END_SEDE, :END_ENTREGA
)

/* sqlInsereDep */
INSERT INTO DEPENDENTE (
  DEPENDENTE.DEP_CODIGO, DEPENDENTE.PES_CODIGO, DEPENDENTE.
  DEP_NOME,
  DEPENDENTE.DEP_DATA_NASCIMENTO
)
VALUES (
  GEN_ID(GEN_DEPENDENTE_ID, 1),
  :PES_CODIGO, :DEP_NOME, :DEP_DATA_NASCIMENTO
)

/* sqlAtualizaForn */
UPDATE PESSOA SET PESSOA.PES_FORNECEDOR = 'V'
WHERE
  PESSOA.PES_CODIGO = :PES_CODIGO

/* sqlAtualizaFunc */
UPDATE PESSOA SET PESSOA.PES_FUNCIONARIO = 'V'
WHERE
  PESSOA.PES_CODIGO = :PES_CODIGO

/* sqlInsereFunc */
INSERT INTO FUNCIONARIO (
  FUNCIONARIO.PES_CODIGO, FUNCIONARIO.FUN_CARGO, FUNCIONARIO.
   FUN_SALARIO
)
VALUES(
  :PES_CODIGO, :FUN_CARGO, :FUN_SALARIO
)
```

Para inserir os dados na tabela pessoa, criamos uma instrução SQL com função de inserção, *sqlInserePessoa*. A mesma instrução servirá para inserir também os funcionários e fornecedores, mudando apenas os parâmetros.

A select contida em *sqlChavePessoa* nos retorna o próximo valor de *Gen_Pessoa_Id*, que foi o *generator* criado para atribuir a chave primária da tabela pessoa. Um *trigger* para atribuir automaticamente o valor ao campo chave, nesse caso, não seria viável, pois a *trigger* é uma rotina que acontece no banco de dados, ou seja, o valor seria visível somente quando a inserção fosse *commitada* (validada e confirmada no banco). Pensando em inserir os endereços e dependentes ao mesmo tempo em que os clientes são importados, veremos mais adiante que é preciso manter esse valor visível em tempo de execução. O *sqlChavePessoa* retorna o valor da chave primária no momento em que é ativado e o mantém até que o componente seja fechado e ativado novamente para trazer um novo valor.

Insira o código da **Listagem 7** no evento *OnClick* do botão Importar Clientes.

Agora podemos rodar o programa e clicar no botão para importar os clientes. É importante lembrar que os *ClientsDataSets* do *dmO* e do *dmD* devem permanecer abertos (*Active*). Caso contrário é necessário ativá-los antes de iniciar a importação dos dados. Observe o resultado na **Figura 16**.

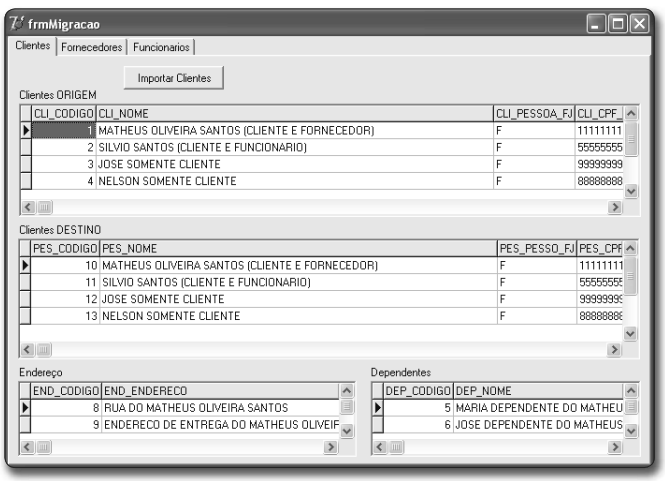


Figura 16. Formulário de Migração com Clientes já importados

Na **Listagem 7**, o comando da **Linha 4** faz com que o *dmO.cdsCliente* aponte para seu primeiro registro. O comando da **Linha 06**, *while not(dmO.cdsCliente.Eof) do*, impõe a condição de que enquanto não for o último registro (*EoF*) de *dmO.cdsCliente*, o compilador deve executa os comandos dentro do corpo (entre *begin* e *end*) e voltar a este ponto. O comando da linha 69 simplesmente aponta para o próximo registro. Ou seja, percorremos todos os registros do arquivo *dmO.cdsCliente*, executando os códigos dentro corpo do comando da **Linha 04** para cada um deles.

Nas **Linhas 09** a **12** estamos renovando o valor da chave primária da tabela pessoa. Se *sqlChavePessoa* já estiver aberto,

então será fechado e aberto novamente. Enquanto não repetirmos esse processo o valor da chave primária (*dmD.sqlChavePessoa.FieldByName('PES_CODIGO').AsInteger*) estará disponível para ser atribuído onde necessário.

Da **Linha 11** a **17** estamos carregando os parâmetros de entrada para executar a instrução sql contida em *sqlInserePessoa*, que insere no bando de dados destino um novo registro na tabela pessoa. Como dito antes, a chave primária não é importada, e sim renovada através do *sqlChavePessoa* na **Linha 11**. Nas próximas linhas são passados os valores dos campos listados em *dmO.cdsCliente*, que contém os dados equivalentes a serem importados. Na **Linha 15** estamos atribuindo o valor equivalente a *verdadeiro* para o campo que define se o registro na tabela pessoa é um cliente. Na **Linha 16** e **17**, informamos que o registro não é fornecedor e nem funcionário. O comando da **Linha 18** executa a instrução SQL, agora já com os devidos parâmetros atribuídos.

Na **Linha 20** é feito um teste para verificar se o cliente possui apenas um endereço (se o endereço de entrega estiver nulo), ou se foram cadastrados dois endereços iguais. Caso contrário significa que o cliente possui um endereço sede e outro para entrega. A instrução SQL contida em *sqlInsereEnd*, insere um novo endereço. Na linha 22, quando a chave do cliente é passada como parâmetro, torna-se claro como é necessário mantermos o valor da chave

Listagem 7. Evento OnClick do botão Importar Clientes.

```
01 procedure Tform1.ImportarClientesClick(Sender: TObject);
02 begin
03   try
04     dmO.cdsCliente.First;
06     while not(dmO.cdsCliente.Eof) do
07     begin
09       if dmD.sqlChavePessoa.Active then
10         dmD.sqlChavePessoa.Close;
11       dmD.sqlChavePessoa.Open;
13       dmD.sqlInserePessoa.Params.ParamByName('PES_CODIGO').
          AsInteger :=
14         dmD.sqlChavePessoa.FieldByName('PES_CODIGO').AsInteger;
15       dmD.sqlInserePessoa.Params.ParamByName('PES_NOME').
          AsString :=
16         dmO.cdsClienteCLI_NOME.AsString;
17       dmD.sqlInserePessoa.Params.ParamByName('PES_PESSO_FJ').
          AsString :=
18         dmO.cdsClienteCLI_PESSOA_FJ.AsString;
19       dmD.sqlInserePessoa.Params.ParamByName('PES_CPF_CNPJ').
          AsString :=
20         dmO.cdsClienteCLI_CPF_CNPJ.AsString;
21       dmD.sqlInserePessoa.Params.ParamByName('PES_CLIENTE').
          AsString := 'V';
22       dmD.sqlInserePessoa.Params.ParamByName('PES_
          FUNCIONARIO').AsString := 'F';
23       dmD.sqlInserePessoa.Params.ParamByName('PES_FORNECEDOR').
          AsString := 'F';
24       dmD.sqlInserePessoa.ExecSQL;
25       if (dmO.cdsClienteCLI_ENDERECO1.AsString =
26         dmO.cdsClienteCLI_ENDERECO2.AsString) or
27       (dmO.cdsClienteCLI_ENDERECO2.IsNull)  then
28       begin
29         dmD.sqlInsereEnd.Params.ParamByName('PES_CODIGO').
            AsInteger :=
30           dmD.sqlChavePessoa.FieldByName('PES_CODIGO').
              AsInteger;
31         dmD.sqlInsereEnd.Params.ParamByName('END_ENDERECO').
            AsString :=
32           dmO.cdsClienteCLI_ENDERECO1.AsString;
33         dmD.sqlInsereEnd.Params.ParamByName('END_NUMERO').
            AsString :=
34           dmO.cdsClienteCLI_NUM_END1.AsString;
35         dmD.sqlInsereEnd.Params.ParamByName('END_BAIRRO').
            AsString :=
36           dmO.cdsClienteCLI_BAIRRO1.AsString;
37         dmD.sqlInsereEnd.Params.ParamByName('END_CEP').AsString :=
38           dmO.cdsClienteCLI_CEP1.AsString;
39         dmD.sqlInsereEnd.Params.ParamByName('END_CIDADE').
            AsString :=
40           dmO.cdsClienteCLI_CIDADE1.AsString;
41         dmD.sqlInsereEnd.Params.ParamByName('END_UF').AsString :=
42           dmO.cdsClienteCLI_UF1.AsString;
43         dmD.sqlInsereEnd.Params.ParamByName('END_SEDE').
            AsString := 'V';
44         dmD.sqlInsereEnd.Params.ParamByName('END_ENTREGA').
            AsString := 'F';
45         dmD.sqlInsereEnd.ExecSQL;
46       end
47       else
48       begin
49         dmD.sqlInsereEnd.Params.ParamByName('PES_CODIGO').
            AsInteger :=
50           dmD.sqlChavePessoa.FieldByName('PES_CODIGO').
              AsInteger;
51         dmD.sqlInsereEnd.Params.ParamByName('END_ENDERECO').
            AsString :=
52           dmO.cdsClienteCLI_ENDERECO1.AsString;
53         dmD.sqlInsereEnd.Params.ParamByName('END_NUMERO').
            AsString :=
54           dmO.cdsClienteCLI_NUM_END1.AsString;
55         dmD.sqlInsereEnd.Params.ParamByName('END_BAIRRO').
            AsString :=
56           dmO.cdsClienteCLI_BAIRRO1.AsString;
57         dmD.sqlInsereEnd.Params.ParamByName('END_CEP').AsString :=
58           dmO.cdsClienteCLI_CEP1.AsString;
59         dmD.sqlInsereEnd.Params.ParamByName('END_CIDADE').
            AsString :=
60           dmO.cdsClienteCLI_CIDADE1.AsString;
61         dmD.sqlInsereEnd.Params.ParamByName('END_UF').AsString :=
62           dmO.cdsClienteCLI_UF1.AsString;
63         dmD.sqlInsereEnd.Params.ParamByName('END_SEDE').
            AsString := 'V';
64         dmD.sqlInsereEnd.Params.ParamByName('END_ENTREGA').
            AsString := 'V';
65         dmD.sqlInsereEnd.ExecSQL;
66         dmD.sqlInsereEnd.Params.ParamByName('PES_CODIGO').
            AsInteger :=
67         dmD.sqlChavePessoa.FieldByName('PES_CODIGO').AsInteger;
68         dmD.sqlInsereEnd.Params.ParamByName('END_ENDERECO').
            AsString :=
69           dmO.cdsClienteCLI_ENDERECO2.AsString;
70         dmD.sqlInsereEnd.Params.ParamByName('END_NUMERO').
            AsString :=
71           dmO.cdsClienteCLI_NUM_END2.AsString;
72         dmD.sqlInsereEnd.Params.ParamByName('END_BAIRRO').
            AsString :=
73         dmO.cdsClienteCLI_BAIRRO2.AsString;
74          dmD.sqlInsereEnd.Params.ParamByName('END_CEP').AsString :=
75           dmO.cdsClienteCLI_CEP2.AsString;
76         dmD.sqlInsereEnd.Params.ParamByName('END_CIDADE').
            AsString :=
77         dmO.cdsClienteCLI_CIDADE2.AsString;
78         dmD.sqlInsereEnd.Params.ParamByName('END_UF').AsString :=
79           dmO.cdsClienteCLI_UF2.AsString;
80         dmD.sqlInsereEnd.Params.ParamByName('END_SEDE').
            AsString := 'F';
81         dmD.sqlInsereEnd.Params.ParamByName('END_ENTREGA').
            AsString := 'V';
82         dmD.sqlInsereEnd.ExecSQL;
93       end;
94       dmO.cdsDependente.First;
95       while not(dmO.cdsDependente.Eof) do
96       begin
97         dmD.sqlInsereDep.Params.ParamByName('PES_CODIGO').
            AsInteger :=
98           dmD.sqlChavePessoa.FieldByName('PES_CODIGO').
              AsInteger;
99         dmD.sqlInsereDep.Params.ParamByName('DEP_NOME').
            AsString :=
100          dmO.cdsDependenteDEP_NOME.AsString;
101        dmD.sqlInsereDep.Params.ParamByName
             ('DEP_DATA_NASCIMENTO').AsDate :=
102          dmO.cdsDependenteDEP_DATA_NASCIMENTO.AsDateTime;
102        dmD.sqlInsereDep.ExecSQL;
103        dmO.cdsDependente.Next;
104      end;
105      dmO.cdsCliente.Next;
106    end;
107 finally
108   dmD.cdsCliente.Close;
109   dmD.cdsCliente.Open;
110 end;
```

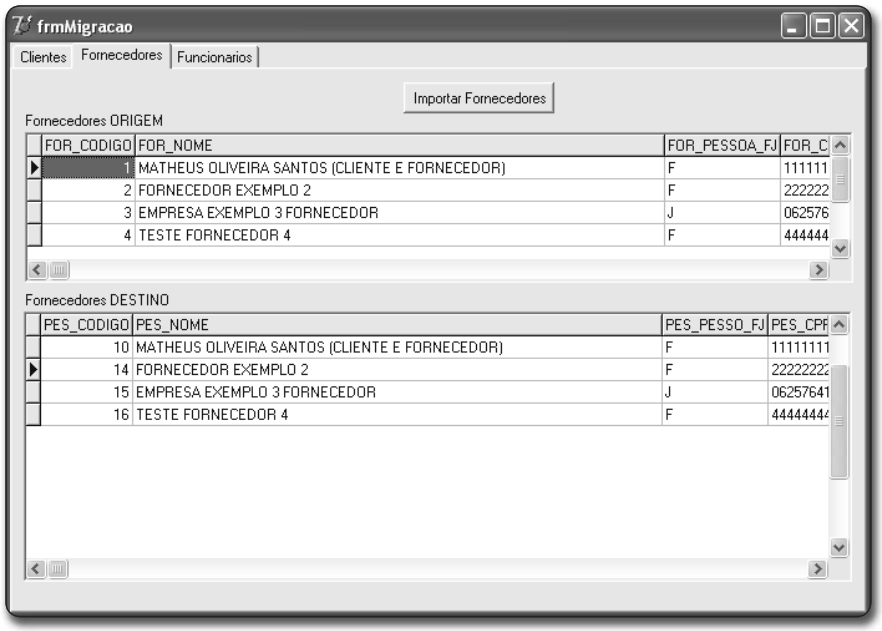


Figura 17. Formulário de Migração com Fornecedores já importados

Listagem 8. Evento OnClick do botão Importar Fornecedores

```
01 procedure Tform1.ImportarFornecedoresClick(Sender: TObject);
02 begin
03 try
04   dmO.cdsFornecedor.First;
05   while not(dmO.cdsFornecedor.Eof) do
06   begin
07     if (dmD.cdsCliente.Locate('PES_CPF_CNPJ',
08       dmO.cdsFornecedorFOR_CPF.AsString, [])) then
09     begin
10       dmD.sqlAtualizaForn.Params.ParamByName('PES_CODIGO').
          AsInteger :=
11         dmD.cdsClientePES_CODIGO.AsInteger;
12       dmD.sqlAtualizaForn.ExecSQL;
13     end
14     else
15     begin
16       if dmD.sqlChavePessoa.Active then
17         dmD.sqlChavePessoa.Close;
18       dmD.sqlChavePessoa.Open;
19       dmD.sqlInserePessoa.Params.ParamByName('PES_CODIGO').
          AsInteger :=
20         dmD.sqlChavePessoa.FieldByName('PES_CODIGO').AsInteger;
21       dmD.sqlInserePessoa.Params.ParamByName('PES_NOME').
          AsString :=
22         dmO.cdsFornecedorFOR_NOME.AsString;
23       dmD.sqlInserePessoa.Params.ParamByName('PES_PESSO_FJ').
          AsString :=
24         dmO.cdsFornecedorFOR_PESSOA_FJ.AsString;
25       dmD.sqlInserePessoa.Params.ParamByName('PES_CPF_CNPJ').
          AsString :=
26         dmO.cdsFornecedorFOR_CPF.AsString;
27       dmD.sqlInserePessoa.Params.ParamByName('PES_CLIENTE').
          AsString := 'F';
28       dmD.sqlInserePessoa.Params.ParamByName('PES_FORNECEDOR').
          AsString := 'V';
29       dmD.sqlInserePessoa.Params.ParamByName
          ('PES_FUNCIONARIO').AsString := 'F';
30       dmD.sqlInserePessoa.ExecSQL;
31       dmD.sqlInsereEnd.Params.ParamByName('PES_CODIGO').
          AsInteger :=
32         dmD.sqlChavePessoa.FieldByName('PES_CODIGO').AsInteger;
33       dmD.sqlInsereEnd.Params.ParamByName('END_ENDERECO').
          AsString :=
34         dmO.cdsFornecedorFOR_ENDERECO.AsString;
35       dmD.sqlInsereEnd.Params.ParamByName('END_NUMERO').
          AsString :=
36         dmO.cdsFornecedorFOR_NUM_END.AsString;
37       dmD.sqlInsereEnd.Params.ParamByName('END_BAIRRO').
          AsString :=
38         dmO.cdsFornecedorFOR_BAIRRO.AsString;
39       dmD.sqlInsereEnd.Params.ParamByName('END_CEP').AsString :=
40         dmO.cdsFornecedorFOR_CEP.AsString;
41       dmD.sqlInsereEnd.Params.ParamByName('END_CIDADE').
          AsString :=
42         dmO.cdsFornecedorFOR_CIDADE.AsString;
43       dmD.sqlInsereEnd.Params.ParamByName('END_UF').AsString :=
44       dmO.cdsFornecedorFOR_UF.AsString;
45       dmD.sqlInsereEnd.Params.ParamByName('END_SEDE').AsString := 'V';
46       dmD.sqlInsereEnd.Params.ParamByName('END_ENTREGA').
          AsString := 'V';
47       dmD.sqlInsereEnd.ExecSQL;
48     end;
49     dmO.cdsFornecedor.Next;
50   end;
51 finally
52   dmD. cdsFornecedor.Close;
53   dmD. cdsFornecedor.Open;
57 end;
```

primária do cliente enquanto não mudamos o registro origem, nesse momento estamos garantindo o vínculo das tabelas.

Da linha 59 até a 67, percorremos todos os registros do *dmO.cdsDependente* e importamos os dados. O *dmO.cdsDependente* mantém um vínculo Mestre-Detalhe com o *dmO.cdsCliente*. Isso significa que sempre que mudar o ponteiro do *dmO.cdsCliente* automaticamente o *dmO.cdsDependente* filtrará seu conteúdo trazendo somente dependentes pertencentes ao cliente atual do *dmO.cdsCliente*.

Nas linhas 72 e 73, ao final de todo o tratamento, o *dmD.cdsCliente* é fechado e ativado novamente para que o *DBGrid* de Clientes destino posso atualizar o seu conteúdo.

A **Listagem 8** mostra o código para ser adicionado no evento *OnClick* do botão Importar Fornecedores. A **Figura 17** mostra o resultado.

Analisando o código da **Listagem 8**, percebemos que, assim como para os clientes, percorremos os registros dos fornecedores para migrá-los um a um.

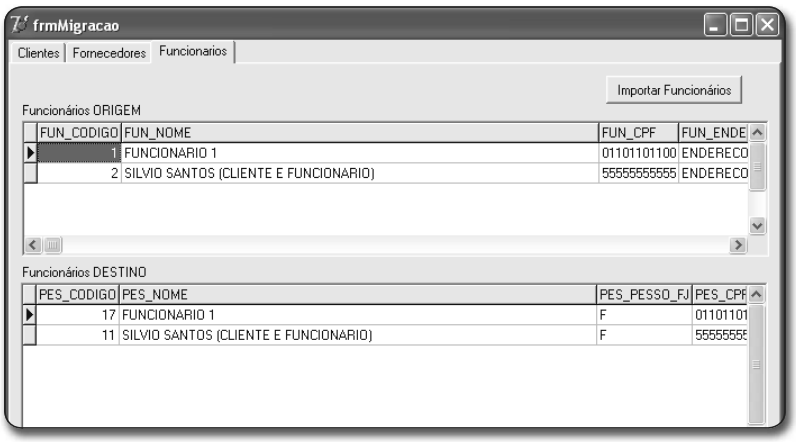


Figura 18. Formulário de Migração com Funcionários já importados

Listagem 9. Evento OnClick do botão Importar Funcionários

```
01 procedure TForm1.ImportarFuncionariosClick(Sender: TObject);
02 begin
03 try
03   dmO.cdsFuncionario.First;
05   while not(dmO.cdsFuncionario.Eof) do
06   begin
07     if (dmD.cdsCliente.Locate('PES_CPF_CNPJ',
08       dmO.cdsFuncionarioFUN_CPF.AsString, [])) then
09     begin
10       dmD.sqlAtualizaFunc.Params.ParamByName('PES_CODIGO').
          AsInteger :=
11         dmD.cdsClientePES_CODIGO.AsInteger;
12       dmD.sqlAtualizaFunc.ExecSQL;
13       dmD.sqlInsereFunc.Params.ParamByName('PES_CODIGO').
          AsInteger :=
14         dmD.cdsClientePES_CODIGO.AsInteger;
15       dmD.sqlInsereFunc.Params.ParamByName('FUN_CARGO').
          AsString :=
16         dmO.cdsFuncionarioFUN_CARGO.AsString;
17       dmD.sqlInsereFunc.Params.ParamByName('FUN_SALARIO').
          AsCurrency :=
18         dmO.cdsFuncionarioFUN_SALARIO.AsCurrency;
19       dmD.sqlInsereFunc.ExecSQL;
20     end
21     else
22     begin
23       if dmD.sqlChavePessoa.Active then
24         dmD.sqlChavePessoa.Close;
25       dmD.sqlChavePessoa.Open;
26       dmD.sqlInserePessoa.Params.ParamByName('PES_CODIGO').
          AsInteger :=
27         dmD.sqlChavePessoa.FieldByName('PES_CODIGO').AsInteger;
28       dmD.sqlInserePessoa.Params.ParamByName('PES_NOME').
          AsString :=
29         dmO.cdsFuncionarioFUN_NOME.AsString;
30       dmD.sqlInserePessoa.Params.ParamByName('PES_PESSO_FJ').
          AsString := 'F';
31       dmD.sqlInserePessoa.Params.ParamByName('PES_CPF_CNPJ').
          AsString :=
32         dmO.cdsFuncionarioFUN_CPF.AsString;
33       dmD.sqlInserePessoa.Params.ParamByName('PES_CLIENTE').
          AsString := 'F';
34       dmD.sqlInserePessoa.Params.ParamByName('PES_FORNECEDOR').
          AsString := 'F';
35       dmD.sqlInserePessoa.Params.ParamByName
          ('PES_FUNCIONARIO').AsString := 'V';
36       dmD.sqlInserePessoa.ExecSQL;
37       dmD.sqlInsereFunc.Params.ParamByName('PES_CODIGO').
          AsInteger :=
38         dmD.sqlChavePessoa.FieldByName('PES_CODIGO').AsInteger;
39       dmD.sqlInsereFunc.Params.ParamByName('FUN_CARGO').
          AsString :=
40         dmO.cdsFuncionarioFUN_CARGO.AsString;
41       dmD.sqlInsereFunc.Params.ParamByName('FUN_SALARIO').
          AsCurrency :=
42         dmO.cdsFuncionarioFUN_SALARIO.AsCurrency;
43       dmD.sqlInsereFunc.ExecSQL;
44       dmD.sqlInsereEnd.Params.ParamByName('PES_CODIGO').
          AsInteger :=
45       dmD.sqlChavePessoa.FieldByName('PES_CODIGO').AsInteger;
46       dmD.sqlInsereEnd.Params.ParamByName('END_ENDERECO').
          AsString :=
47         dmO.cdsFornecedorFOR_ENDERECO.AsString;
48       dmD.sqlInsereEnd.Params.ParamByName('END_NUMERO').
          AsString :=
49         dmO.cdsFornecedorFOR_NUM_END.AsString;
50       dmD.sqlInsereEnd.Params.ParamByName('END_BAIRRO').
          AsString :=
51         dmO.cdsFornecedorFOR_BAIRRO.AsString;
52       dmD.sqlInsereEnd.Params.ParamByName('END_CEP').AsString :=
53         dmO.cdsFornecedorFOR_CEP.AsString;
54       dmD.sqlInsereEnd.Params.ParamByName('END_CIDADE').
          AsString :=
55         dmO.cdsFornecedorFOR_CIDADE.AsString;
56       dmD.sqlInsereEnd.Params.ParamByName('END_UF').AsString :=
57         dmO.cdsFornecedorFOR_UF.AsString;
58      dmD.sqlInsereEnd.Params.ParamByName('END_SEDE').AsString := 'V';
59       dmD.sqlInsereEnd.Params.ParamByName('END_ENTREGA').
          AsString := 'V';
59       dmD.sqlInsereEnd.ExecSQL;
60     end;
61     dmO.cdsFuncionario.Next;
62   end;
63 finally
64   dmD.cdsFuncionario.Close;
65   dmD.cdsFuncionario.Open;
66 end;
```

A **Linha 06** faz um teste para ver se já existem um cliente com o mesmo CPF do fornecedor a ser incluído. Caso haja, será executada a instrução SQL que apenas atualiza o campo *Pes_Fornecedor* da tabela *Pessoa* com o valor que representa *verdadeiro*. É passado como parâmetro a chave primária do registro que foi localizado.

Da **Linha 13** a **36** será feita a inserção de um novo registro na tabela pessoa, caso não passe pelo teste da **Linha 06**. É o mesmo processo para inserir clientes, só mudando os parâmetros, que dessa vez recebe os dados do *dmO.cdsFornecedor*. Também é configurado o tipo de pessoa nas **Linhas 21** a **23**.

Repare no exemplo da **Figura 17**, que o fornecedor de código 10 é na verdade o mesmo registro que aparece como cliente de código 10 na **Figura 16**.

A **Listagem 9** mostra o código a ser inserido no botão Importar Fornecedores. A **Figura 18** mostra o resultado.

Como pode ser visto na **Listagem 9**, percorremos o *dmO.cdsFuncionario* e importamos os dados. Na **Linha 06** é feito um teste para conferir ser há um cliente com o mesmo CPF do atual funcionário. Caso haja, é executada a instrução SQL que

simplesmente atualiza o cliente encontrado, marcando-o como pessoa também do tipo funcionário. Aproveitamos o registro encontrado para passar sua chave primária como parâmetro para a instrução SQL que insere um novo registro na tabela auxiliar funcionario, *sqlInsereFunc*. Caso não encontre um cliente, é inserido um novo registro na tabela pessoa configurado como sendo somente funcionário (pes_funcionario = 'V').

Repare no exemplo da **Figura 18**, que o funcionário de código 11 é na verdade o mesmo registro que aparece como cliente de código 11 na **Figura 16**.

Conclusão

Um desenvolvedor de software deve estar sempre preparado para encontrar obstáculos no decorrer do seu trabalho. Migrar dados de um banco para outro é apenas um deles e, com certeza, a aplicação construída neste artigo não é a solução para todas as situações em que um programador precisa substituir um banco de dados. Mas construindo um exemplo na prática, o programador iniciante exercita sua capacidade de solucionar problemas que ocorrem em diferentes condições relacionadas ao assunto. Através deste exemplo vimos como uma aplicação em Delphi pode nos ajudar a tratar as diferenças, impor condições para se manter a integridade dos dados e garantir que o sistema continue a usar os antigos registros com segurança. ●

Dê seu feedback sobre esta edição!

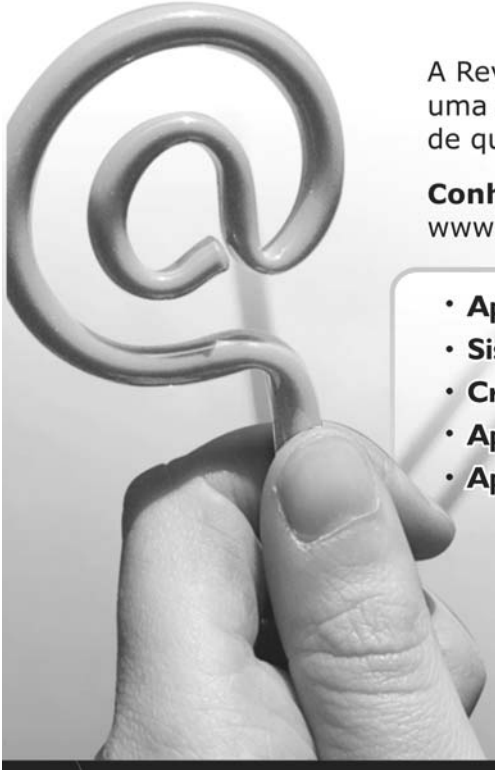
A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Cursos Online

Assinatura ClubeDelphi PLUS
Mais conteúdo DELPHI por muito menos!



A Revista **Clubedelphi** oferece para seus assinantes uma série de Cursos Online de alto padrão de qualidade .

Conheça abaixo os cursos já disponíveis:
www.devmedia.com.br/curso/cdplus

- **Aplicações Client/Server com dbExpress e Firebird**
- **Sistema SysCash**
- **Criando uma aplicação multi-camadas completa com Delphi**
- **Aplicações client/server com Windows Forms no Delphi 2006**
- **Aplicações WEB com IntraWeb e Delphi 7 (Delphi Win32)**

A sua melhor opção de aprendizagem!

Assine a **Clubedelphi** e
Comece já seu treinamento!
www.devmedia.com.br/assine

DevMedia GROUP Mais Informações: www.devmedia.com.br/central - Tel.: 21 3382-5038/ 3382-5025

Nesta seção você encontra artigos intermediários sobre Delphi Win 32 e Delphi for .NET

Transporte de dados em multicamadas e DataSnap

Saiba como enviar e receber dados das aplicações Cliente e Servidora

Nesse artigo veremos

- Criação do banco de dados de exemplo;
- Criação de aplicação multicamadas utilizando BSS;
- Interação entre os componentes ClientDataSet e DataSetProvider.

Qual a finalidade

- Realizar um processo de interação entre as camadas, garantindo maior facilidade de manutenção e agilidade para implementação de determinados processos;
- Permitir o envio de dados extra, utilizando os eventos presentes na tecnologia DataSnap.

Quais situações utilizam esse recurso?

Qualquer aplicação desenvolvida em arquitetura multicamadas e tecnologia DataSnap, pode desfrutar destes recursos.

Resumo do DevMan

Desenvolvedores que comumente estão acostumados com o ambiente Cliente/Servidor e optam por realizar a migração de seus softwares para o ambiente multicamadas geralmente encontram dificuldades em definir certos processos.

Porém, quem já está um pouco mais ambientado com esta arquitetura sabe da necessidade de se trafegar dados extras, informações que muitas vezes você já possui, mas por não saber como trafegá-las para seu servidor de aplicação, opta por obtê-las diretamente no servidor.

No presente artigo procuro demonstrar algumas formas de interação entre as camadas, através da tecnologia DataSnap, possibilitando enviar informações extras da aplicação cliente para a servidora e vice-versa.

Daniel Sobrinho Laporte
daniel.laporte@gmail.com
Editor técnico da revista ClubeDelphi. Graduado em sistemas de informação e pós-graduado em administração de bancos de dados pela Universidade de Franca. É analista de sistemas e desenvolvedor Delphi desde 2005 desenvolvendo aplicações cliente/servidor e multicamadas em Delphi com Firebird.

Há pouco tempo atrás na empresa onde atualmente trabalho, foi necessário criar um determinado cadastro que ao ser alimentado gerava ocorrências em outras tabelas que não tinham relação nenhuma entre si. Nesta situação eram necessários alguns cálculos complexos e algumas outras validações antes de concluir este processo. Em um ambiente cliente-servidor eu poderia muito bem utilizar algumas *triggers*, realizar algumas pequenas adaptações e todo este problema estaria resolvido facilmente, centralizando assim todas as ações no meu banco de dados.

Porém, uma das opções de quem privilegia a utilização da arquitetura

N-Tier, é optar pela carga realizada no servidor de aplicação e utilizar o banco de dados com o mínimo de objetos possíveis dentro de si. Desta forma, qualquer futura migração para outro banco de dados poderá ser realizada sem que haja grandes dores de cabeça.

Para quem já trabalha neste ambiente, certamente está acostumado a trafegar dados pela rede utilizando o DataSnap. Para isso, geralmente criamos métodos no servidor de aplicação e de alguma forma, passamos algum parâmetro necessário para sua alimentação. Porém, existem maneiras de se trafegar dados pela rede aproveitando os recursos presentes na tecnologia *DataSnap*, permitidas através das interações entre os eventos dos componentes *ClientDataSet* e *DataSetProvider*. Neste artigo, veremos como utilizar alguns destes eventos para realizar diversas tarefas que podem agilizar os processos de atualização de dados, inclusões, obtenção de informações do servidor de aplicação, entre outras tantas possibilidades

Compreendo a interatividade entre os componentes

O interessante da tecnologia *DataSnap* é justamente a comunicação e a interatividade entre o *ClientDataSet* e o *DataSetProvider*. Se repararmos nos eventos dos dois componentes, veremos que uma boa parte dos eventos do *ClientDataSet* também estão presentes no *DataSetProvider*. O que acontece é que inicialmente os eventos são chamados na aplicação Cliente (onde fica o *ClientDataSet*), que encarrega-se de gerar a requisição ao servidor. Já no servidor (onde fica o *DataSetProvider*), os dados são recebidos e todo o processo de atualização, validação e persistência é realizado. O resultado, que pode ser um evento de erro ou alguma informação extra, é retornado à aplicação solicitante (Cliente). Todo este processo é detalhado na **Figura 1**.

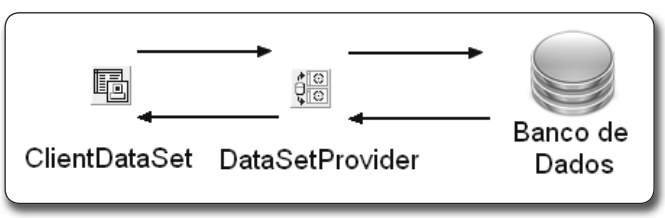


Figura 1. Conectividade entre ClientDataSet e DataSetProvider

Criando o banco de dados de exemplo

Para o presente modelo, pretendo apresentar um banco de dados com apenas duas tabelas que não possuem interatividade nenhuma entre si. Em nosso exemplo utilizarei o banco de dados *Firebird* em sua versão *1.5*, porém, este exemplo é perfeitamente aplicável a qualquer outro banco de dados ou versão de sua preferência.

Conforme podemos ver na **Figura 2,** apenas temos a tabela *CLIENTE*, com informações simples sobre o cliente e a tabela *LOG* que possui apenas o código sequencial (*ID*) juntamente com o campo descritivo da ação (*LOG*). Perceba também que estas não possuem absolutamente nenhum relacionamento entre si. O *script* para a criação do banco de dados de exemplo está descrito conforme mostrado na **Listagem 1**.

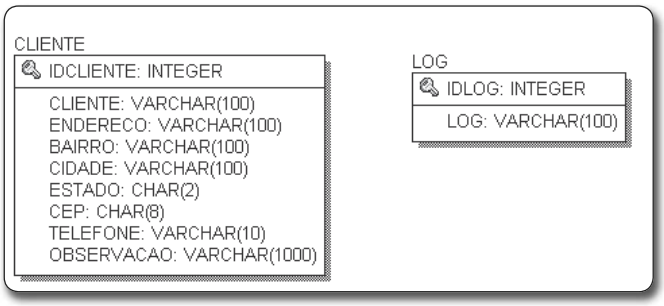


Figura 2. Modelagem do banco de dados

Listagem 1. Script do banco de dados

```
SET SQL DIALECT 3;
SET NAMES WIN1252;

SET CLIENTLIB '<DIRETÓRIO DA INSTALAÇÃO DO FIREBIRD>\BIN\
fbclient.dll';

CREATE DATABASE '<DIRETÓRIO DO BANCO DE DADOS>\EXEMPLO.FDB'
USER 'SYSDBA' PASSWORD 'masterkey'
PAGE_SIZE 4096
DEFAULT CHARACTER SET WIN1252;

CREATE TABLE CLIENTE (
    IDCLIENTE   INTEGER NOT NULL,
    CLIENTE     VARCHAR(100) NOT NULL,
    ENDERECO    VARCHAR(100),
    BAIRRO      VARCHAR(100),
    CIDADE      VARCHAR(100),
    ESTADO      CHAR(2),
    CEP         CHAR(8),
    TELEFONE    VARCHAR(10),
    OBSERVACAO  VARCHAR(1000)
);

ALTER TABLE CLIENTE ADD CONSTRAINT
PK_IDCLIENTE PRIMARY KEY (IDCLIENTE)
USING INDEX IDX_CLIENTE;

CREATE TABLE LOG (
    IDLOG  INTEGER NOT NULL,
    LOG    VARCHAR(100)
);

ALTER TABLE LOG ADD CONSTRAINT PK_IDLOG PRIMARY KEY (IDLOG)
USING INDEX IDX_IDLOG;

CREATE GENERATOR GEN_IDCLIENTE;
CREATE GENERATOR GEN_IDLOG;
```

Nota do DevMan

Em computação, Log de dados é o termo utilizado para descrever o processo de registro de eventos relevantes num sistema computacional. Esse registro pode ser utilizado para reestabelecer o estado original de um sistema ou para que um administrador conheça o seu comportamento no passado. Um arquivo de log pode ser utilizado para auditoria e diagnóstico de problemas em sistemas computacionais.

Fonte: Wikipédia.

Desenvolvendo a aplicação servidora

Iniciaremos o nosso exemplo criando a aplicação servidora. Para este exemplo estarei utilizando o *RAD Studio 2007*, mas devo salientar que é perfeitamente possível aplicar estas práticas a outras versões. Abra seu *RAD Studio 2007* e vá em *File>New>VCL Forms Application – Delphi for Win32* para iniciar uma nova aplicação, assim sendo, altere suas propriedades *Name* para *frmServidor* e *Caption* para *Servidor de Aplicações BSS*, em seguida altere a cor do formulário através da propriedade *Color* setando-a para *clBlack*,

redimensione o formulário para 165 (*Heigth*) e 266 (*Width*). Adicione ao novo formulário três componentes *TLabel*, (*lblDescricao, lblData, lblHora*) com o *Caption* respectivamente definido como *Servidor BSS, Data, Hora*. Agora insira um componente *Timer (TTimer)* e em seu evento *OnTimer* adicione o código a seguir para que constantemente a data e hora sejam atualizados no servidor.

```
lblData.Caption := 'Data : ' + FormatDateTime('mm/dd/yyyy', Date);
lblHora.Caption := 'Hora : ' + FormatDateTime('hh:nn:ss', Now);
```

Salve seu projeto, no menu *File>Save All*, nomeie a *Unit* como *untServidor.pas* e o projeto como *AppServerD2007.dprj*. Agora iremos criar o *Remote Data Module* que possibilitará a comunicação com a camada cliente. Para isso vá ao menu *File>New>Other.>Multitier* e do lado direito selecione o *Remote Data Module*. Na janela de *wizard* crie-o informando em *CoClass Name* a descrição de *RDMPrincipal*, defina *Instancing* como *Multiple Instance* e *Threading Model* como *Single* conforme ilustra a **Figura 3.**

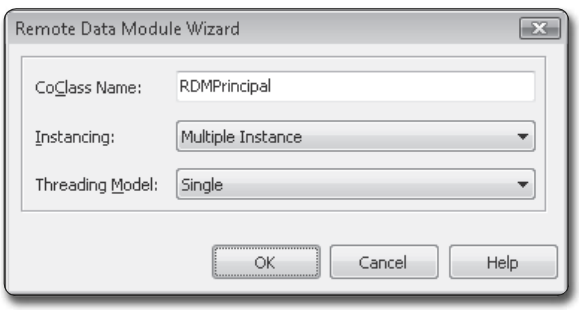


Figura 3. Remote Data Module Wizard

No novo *Remote Data Module* adicione um *SQLConnection* (*Conexao*), quatro *SQLDataSet's* (*sqlCliente, sqlAuxiliar, sqlGetID, sqlLog*) e dois *DataSetProvider's* (*dspCliente, dspAuxiliar*). Selecione o componente *Conexao* e com um duplo clique crie uma nova conexão, utilizando o *Driver* do *Interbase*, conforme mostra a **Figura 4.**

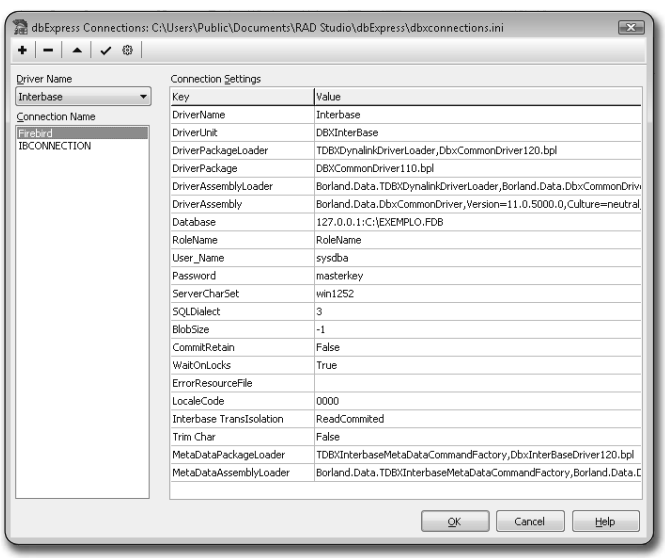


Figura 4. Criação da conexão com o banco de dados

Após ter criado a conexão com o banco de dados, ainda no componente *Conexao*, altere a propriedade *LoginPrompt* para *False* e *VendorLib* para *fbclient.dll*. Selecione agora todos os *SQLDataSet's* e informe à propriedade *SQLConnection* o componente *Conexao* e em seguida digite na propriedade *CommandText* as instruções *SQL* respectivas de cada componente, conforme mostra a **Listagem 2.**

Listagem 2. Sentenças SQL dos SQLDataSet's

```
[sqlCliente]
SELECT
  CLIENTE.IDCLIENTE,
  CLIENTE.CLIENTE,
  CLIENTE.ENDERECO,
  CLIENTE.BAIRRO,
  CLIENTE.CIDADE,
  CLIENTE.ESTADO,
  CLIENTE.CEP,
  CLIENTE.TELEFONE,
  CLIENTE.OBSERVACAO
FROM
  CLIENTE
WHERE
  CLIENTE.IDCLIENTE =:IDCLIENTE

[sqlLog]
INSERT INTO LOG
( IDLOG,
  LOG )
VALUES
((SELECT GEN_ID(GEN_IDLOG, 1) FROM RDB$DATABASE),
 :LOG )
```

Observando o código da **Listagem 2** referente ao *sqlLog*, podemos perceber que a instrução da obtenção da chave será realizada automaticamente, pois já possui a instrução de obtenção do valor do *Generator* no banco de dados, deste modo teremos apenas que informar a descrição da ocorrência.

Vamos seguir realizando as configurações necessárias. Selecione primeiramente o *sqlCliente* e dê um clique sobre a propriedade *Params* para que o editor de parâmetros seja aberto. Selecione o parâmetro *IDCLIENTE* e configure sua propriedade *DataType* para *ftInteger*. Agora dê um duplo clique sobre o *sqlCliente* para que o *Fields Editor* seja exibido, clique com o botão direito sobre ele e selecione a opção *Add all fields* para que todos os campos sejam adicionados a ele. Selecione um a um e configure suas propriedades *ProviderFlags* de forma que fiquem como *True* as opções [*pfInUpdate, pfInKey, pfInWhere*] para a chave primária (*IDCLIENTE)* e apenas [*pfInUpdate*] para os campos que são pertencentes a mesma tabela e deverão ser atualizados (todos os demais campos). Modifique também a propriedade *Required* do campo *IDCLIENTE*, para que a validação seja feita pelo *DataSsetProvider* no momento da persistência dos dados. Feito isso, selecione o *sqlLog* e novamente clique sobre a propriedade *Params*, porém, desta vez configure o parâmetro *LOG* com *DataType ftString* para que assim sejam finalizadas as configurações dos *SQLDataSet's*.

ClubeDelphi PLUS www.devmedia.com.br/clubedelphi/portal.asp

Acesse agora mesmo o portal do assinante ClubeDelphi e assista a uma vídeo aula de Everson Borges Volaco que mostra como criar campos auto-incremento utilizando Generators do Firebird.

*http://www.devmedia.com.br/articles/viewcomp.asp?comp=1070&hl=*Generator**

Partindo para a configuração dos *DataSetProvider's* faça primeiramente a ligação entre o *dspCliente* e o *sqlCliente*. Ligue o *sqlCliente* na propriedade *DataSet* do *dspCliente*, ainda nele, habilite as *Options* [*poIncFieldProps, poPropagateChanges*] para que todas as alterações nos campos, sejam automaticamente propagadas para o *ClientDataSet*. Finalizando este *DataSetProvider,* apenas altere sua propriedade *UpdateMode* para *upWhereKeyOnly.* O componente *dspAuxiliar* deve apenas ser ligado ao *sqlAuxiliar* através da propriedade *DataSet.* Com isso finalizamos a configuração dos componentes do servidor de aplicação. Um exemplo do seu *Remote Data Module* já finalizado pode ser visto na **Figura 5.**

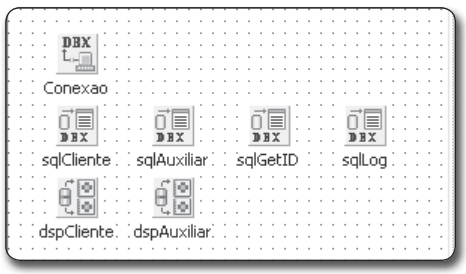


Figura 5. Criação do Remote Data Module de exemplo

Agora que concluímos o processo de configuração passaremos para a codificação necessária. Antes de qualquer coisa, adicione a *Unit Variants* à *uses* do seu *RDMPrincipal* e em seguida implemente o evento *OnDataRequest* no componente *dspAuxiliar* conforme mostra a **Listagem 3.**

O evento *OnDataRequest* já faz referência ao tema deste artigo. Este comumente é utilizado para consultas ou algum outro tipo de requisição como o próprio nome sugere. Sua ocorrência se dá, quando o *ClientDataSet* utiliza o método *DataRequest* passando como parâmetros uma estrutura de dados do tipo *OleVariant*, que nada mais é, do que um *Array* de dados do tipo *Variant.* Analisando a **Listagem 3** podemos notar entre as **Linhas 04** e **20,** a declaração de duas constantes (*sqlBusca_Cliente, sql_Busca_Log*) que contém as instruções de busca de informações referentes as duas tabelas do nosso banco de dados. Perceba também que estamos utilizando o mesmo componente para realizar busca em duas tabelas e não utilizamos a propriedade *CommandText* do *ClientDataSet.* Seguindo, temos a limpeza da propriedade *CommandText* pertencente ao *sqlAuxiliar* (**Linha 22**), e adiante uma sequência de comandos *case.* O primeiro deles (**Linha 23**), trata qual será a tabela que será efetuada a busca, caso o valor (recebido por parâmetro através da *Array* de *Variants Input*, seguido de sua posição) seja *0,* será selecionada a consulta optando pela tabela de *Clientes,* caso o valor seja *1* a consulta será realizada utilizando a tabela de *Log's.* O próximo *case* da opção de busca por clientes, trata como podemos ver a posição *1* do parâmetro *Input*. Contendo o valor *0,* a busca será realizada passando o valor de *Input* na posição *2* para selecionar pelo *ID* do cliente, caso contrário, será utilizado o nome do cliente para consulta (**Linhas 27** à **31**). O mesmo ocorre para a tabela de *Log's*, porém, tratando a busca pelo *IDLOG* ou pela própria descrição do log (**Linhas 37** à **41**). Mais adiante entenderemos melhor, na construção da nossa aplicação cliente.

Listagem 3. Evento OnDataRequest do dspAuxiliar

```
01  function TRDMPrincipal.dspAuxiliarDataRequest(Sender: TObject;
02    Input: OleVariant): OleVariant;
03  const
04    sql_Busca_Cliente =
05      ' SELECT '              + #13 +
06      ' CLIENTE.IDCLIENTE, ' + #13 +
07      ' CLIENTE.CLIENTE, '   + #13 +
08      ' CLIENTE.ENDERECO, '  + #13 +
09      ' CLIENTE.BAIRRO, '    + #13 +
10      ' CLIENTE.CIDADE, '    + #13 +
11      ' CLIENTE.ESTADO, '    + #13 +
12      ' CLIENTE.CEP, '       + #13 +
13      ' CLIENTE.TELEFONE, '  + #13 +
14      ' CLIENTE.OBSERVACAO ' + #13 +
15      ' FROM CLIENTE ';
16    sql_Busca_Log =
17      ' SELECT '     + #13 +
18      ' LOG.IDLOG, ' + #13 +
19      ' LOG.LOG '    + #13 +
20      ' FROM LOG ';
21  begin
22    sqlAuxiliar.CommandText := '';
23    case Input[0] of
24      0:
25      begin
26        sqlAuxiliar.CommandText := sql_Busca_Cliente;
27        case Input[1] of
28          0: sqlAuxiliar.CommandText := sqlAuxiliar.CommandText +
29            ' WHERE IDCLIENTE = ' + Input[2];
30          1: sqlAuxiliar.CommandText := sqlAuxiliar.CommandText +
31            ' WHERE CLIENTE LIKE ' + QuotedStr(Input[2]) + ' || ' +
                    QuotedStr('%');
32        end;
33      end;
34      1:
35      begin
36        sqlAuxiliar.CommandText := sql_Busca_Log;
37        case Input[1] of
38          0: sqlAuxiliar.CommandText := sqlAuxiliar.CommandText +
39            ' WHERE IDLOG = ' + Input[2];
40          1: sqlAuxiliar.CommandText := sqlAuxiliar.CommandText +
41            ' WHERE LOG LIKE ' + QuotedStr(Input[2]) + ' || ' +
              QuotedStr('%');
42        end;
43      end;
44    end;
45  end;
```

O próximo passo é configurarmos as ações restantes designadas ao nosso servidor de aplicação. Selecione o componente *dspCliente* e no evento *BeforeUpdateRecord*, adicione o código da **Listagem 4.**

Na **Listagem 4** temos a segunda interação entre o *ClientDataSet* e o *DataSetProvider* como iremos perceber. Na **Linha 09** é verificado se existe algum conteúdo no campo *IDCLIENTE.* Caso não haja nenhum tipo de valor, o componente *sqlGetID* realizará uma instrução de busca no banco de dados, obtendo o valor do *Generator* referente ao campo chave e logo em seguida atribuindo ao conteúdo que anteriormente estava nulo (**Linhas 11** à **15**). Na **Linha 17,** perceba que verificamos se há conteúdo em um parâmetro opcional, denominado de *PARAMETRO*. Caso esta condição seja verdadeira, o campo observação receberá o conteúdo deste parâmetro (**Linha 18**). Por último, o *log* é alimentado de acordo com o tipo de ação que está sendo realizada, verificando-se o *UpdateKind*. Podemos ver que a variável *stStatus* é alimentada com o conteúdo da ação e passada como parâmetro para o *Log* (**Linhas 17** a **23**).

Nota

Qualquer erro que ocorra, todo o processo será automaticamente abortado simulando o efeito de um Rollback.

Listagem 4. Evento BeforeUpdateRecord do dspCliente

```
01  procedure TRDMPrincipal.dspClienteBeforeUpdateRecord
     (Sender: TObject;
02    SourceDS: TDataSet; DeltaDS: TCustomClientDataSet;
       UpdateKind:
03    TUpdateKind; var Applied: Boolean);
04  var
05    stStatus : String;
06  begin
07    if SourceDS = sqlCliente then
08    begin
09      if VarIsEmpty(DeltaDs.FieldByName('IDCLIENTE').NewValue) then
10      begin
11        sqlGetID.Close;
12        sqlGetID.CommandText := '';
13        sqlGetID.CommandText := ' SELECT GEN_ID(GEN_IDCLIENTE,1) FROM
            RDB$DATABASE ';
14        sqlGetID.Open;
15        DeltaDS.FieldByName('IDCLIENTE').NewValue :=
            sqlGetID.Fields[0].Value;
16      end;
17      if not VarIsEmpty(DeltaDs.GetOptionalParam('PARAMETRO')) then
18        DeltaDs.FieldByName('OBSERVACAO').NewValue :=
            DeltaDs.GetOptionalParam('PARAMETRO');
19      sqlLog.Close;
20      case UpdateKind of
21      ukModify : stStatus := 'MODIFICADO REGISTRO NA TABELA DE
         CLIENTES';
22      ukInsert : stStatus := 'INSERIDO REGISTRO NA TABELA DE
         CLIENTES';
23      ukDelete : stStatus := 'DELETADO REGISTRO NA TABELA DE
         CLIENTES';
24      end;
25      sqlLog.Params[0].AsString := stStatus;
26      sqlLog.ExecSQL;
27  end;
```

Para finalizar o servidor de aplicação apenas adicione o código a seguir no evento *AfterApplyUpdates* do *dspCliente.*

```
procedure TRDMPrincipal.dspClienteAfterApplyUpdates(Sender: TObject;
  var OwnerData: OleVariant);
begin
  OwnerData := VarArrayOf([FormatDateTime('hh:nn:ss',Time)]);
end;
```

Veja que na declaração do método existe um parâmetro chamado *OwnerData* do tipo *OleVariant.* Este parâmetro permite que seja passado algum conteúdo do *DataSetProvider* no servidor de aplicação para o *ClientDataSet* da aplicação cliente. Neste caso, iremos retornar na aplicação cliente a hora em que ocorreu a gravação dos dados no banco de dados. Lembrando que caso haja algum erro no *BeforeUpdateRecord* do mesmo componente, este evento não ocorrerá.

Com isso finalizamos o nosso servidor de aplicação, salve seu projeto nomeando a *Unit* como *untRDMPrincipal.pas.*

Preparando o ambiente

Antes de passarmos para o desenvolvimento da aplicação cliente, quero realizar algumas observações.

O primeiro passo é registrar o servidor de aplicação no seu sistema operacional. Para isso, acesse o menu *Project>Options>Debugger* e em *Parameters* adicione a instrução */regserver* e clique em *Ok.* Agora simplesmente execute o seu projeto (*F9*), para que seu servidor de aplicação esteja corretamente registrado no sistema operacional.

Nota

Lembre-se de executar esta instrução apenas no momento de registrar o servidor de aplicação. Caso necessite realizar o debug do projeto, é importante que este parâmetro seja removido, caso contrário não será possível realizar este procedimento.

O segundo passo é iniciar o executável do *Borland Socket Server.* Para isso vá no diretório *bin* da instalação do seu *RAD Studio 2007* e dê um duplo clique sobre o executável *scktsrvr.exe.*

Nota

Como estamos em ambiente de desenvolvimento, é altamente recomendado que o scktsrvr. exe seja executado como aplicação e não como serviço. Caso instale-o como serviço, não será possível realizar o debug de sua aplicação servidora. Para instalar o Borland Socket Server como um serviço, simplesmente execute-o no prompt de comando com o parâmetro - install. Para desinstalar basta realizar o mesmo processo, porém com o processo - uninstall.

Desenvolvendo a aplicação cliente

Após ter realizado os passos anteriormente descritos, inicie uma nova aplicação através do menu *File>New> VCL Forms Application - Delphi for Win32*. Em seguida crie um novo *Data Module* através do menu *File>New>Other> Delphi Files* e do lado direito selecione o *Data Module.* Renomeie-o para *DM-Principal*. Antes de inserir os componentes adicione à cláusula *uses* do seu *Data Module* as *Unit's Variants* e *Dialogs.* Feito isso, insira um componente *SocketConnection* (*sckPrincipal*) e dois *ClientDataSet's (cdsCliente, cdsAuxiliar).* Primeiramente selecione o *sckPrincipal* e informe o endereço de *IP* onde estará o servidor de aplicação. Em nosso caso utilizei o endereço local: 127.0.0.1. Aponte agora à sua propriedade *ServerName* o *AppServerD2007. RDMPrincipal* para que a ligação esteja realizada. Selecione em seguida os dois *ClientDataSet's* e configure à suas propriedades *RemoteServer* o *sckPrincipal*, e selecione os respectivos *DataSetProvider's* através da propriedade *ProviderName.* Por último dê um duplo clique sobre o *cdsCliente* e adicione todos os campos, selecionando *Add all fields*, para que as configurações estejam finalizadas. Ainda no componente *cdsCliente* codifique o evento *AfterApplyUpdates* conforme código a seguir:

```
procedure TDMPrincipal.cdsClienteAfterApplyUpdates(Sender: TObject;
  var OwnerData: OleVariant);
begin
  ShowMessage('Hora da atualização no servidor : ' +
    VarToStr(Ownerdata[0]));
end;
```

Como anteriormente dito, na construção da aplicação servidora o *DataSetProvider* se encarregará de enviar a hora da atualização no servidor de aplicação, deste modo, o *cdsCliente* trata o parâmetro *OwnerData* em sua posição *0*, convertendo-o para um tipo *String* e exibindo a mensagem após a atualização. Após ter concluído o seu *Data Module*, salve seu projeto (*File>Save All*), atribua os nomes de *untAppCliente.pas* ao formulário e *untDMPrincipal.pas* ao *Data Module*. O projeto deverá ser salvo com o nome de *AppCliente.dprj.*

Passaremos agora para o formulário de controle que será responsável por realizar a interação com o formulário principal da aplicação. Crie um novo formulário acessando *File>New>Form – Delphi for win32*. Altere suas propriedades *Name* para *frmControle, Caption* para *Informe* e *Position* para *poMainFormCenter.* Adicione agora um *PageControl* (*pclPrincipal*), clique sobre ele com o botão

direito e adicione duas páginas (*New Page*). Configure a primeira com o *Name* de *tbsOnDataRequest* e a segunda, *tbsOptionalParam*. Na aba *tbsOnDataRequest* adicione três *Label's* (*Cliente, Campo, Conteúdo*), dois controles *ComboBox* (*cmbTabela, cmbCampo*) e defina suas propriedades *Style* como *csOwnerDrawFixed*. Para o *cmbTabela*, adicione à *Items (Cliente, Log)*, já no *cmbCampo* adicione (*Código do Cliente, Cliente*) e por último adicione um Edit (*edtConteud*). Agora na aba *tbsOptionalParam*, adicione um *Label* (*Conteúdo do Parâmetro*) e um *Memo* (*memParametros*). Por último, adicione ao formulário um *Panel* (*pnlRodape*) e dentro deste, dois *BitBtn's* (*btnOk, btnCancelar*). No *btnOk* modifique a propriedade *ModalResult* para *mrOk* e defina *mrCancel* para a mesma propriedade no *btnCancelar*. Finalize configurando a propriedade *Align* do *pclPrincipal* para *alClient*, e *alBottom* para o *pnlRodape*.

Finalizando o nosso artigo iremos configurar o formulário principal de nossa aplicação que anteriormente havia sido criado, mas não modificado. Modifique seu *Name* para *frmAppCliente* e seu *Caption* para *Aplicação de Teste de Transporte de Dados*. Novamente antes de inserir componentes, informe à cláusula *uses* do seu *Form* as *Unit's DBClient, untPrincipal* e *untControle*. Adicione dois componentes *DataSource* (*dsrPrincipal, dsrPesquisa*), dois *Panel's* (*pnlAcoes, pnlPrincipal*), e dois *BitBtn's* (*btnRequest, btnApply*), em seguida adicione um *DBMemo*. No *DBMemo* altere para *True* a propriedade *ReadOnly* e *clBtnFace* para a propriedade *Color*. Continue a adição de componentes, inserindo um *DBNavigator*, um *DBText*, oito *Label's* e sete *DBEdit's*. Ligue a propriedade *DataSource* do *DBGrid* ao *dsrPesquisa* e os demais componentes *DB-Aware* ao *dsrPrincipal*. O resultado final da nossa interface pode ser observado na **Figura 6.**

Partiremos agora para a codificação. Inicie criando duas propriedades na sessão *public* do seu formulário, conforme mostra o código seguinte, em seguida pressione as teclas *Ctrl + Shift + C* para que os métodos sejam criados automaticamente.

```
property DataSetPrincipal : TClientDataSet;
property DataSetPesquisa  : TClientDataSet;
```

Após ter feito isso, duas *procedures* serão criadas. Modifique a *setDataSetPrincipal* para:

```
FDataSetPrincipal := Value;
dsrPrincipal.DataSet := FDataSetPrincipal;
```

Em seguida modifique o método *setDataSetPesquisa* como mostrado a seguir:

```
FDataSetPesquisa := Value;
dsrPesquisa.DataSet := FDataSetPesquisa;
```

No evento *OnCreate* do seu formulário adicione o seguinte código:

```
DataSetPrincipal := DMPrincipal.cdsCliente;
DataSetPesquisa  := DMPrincipal.cdsAuxiliar;
DataSetPrincipal.Open;
```

Nota

A implementação deste formulário não será mostrada para que o artigo não seja muito estendido, porém, o download do exemplo se encontra disponível no site da DevMedia.

Nota do DevMan

O IP (Internet Protocol ou Protocolo de Internet), de forma genérica, pode ser considerado como um conjunto de números que representam o local de um determinado equipamento (normalmente computadores) em uma rede privada ou pública.

Fonte: Wikipédia.

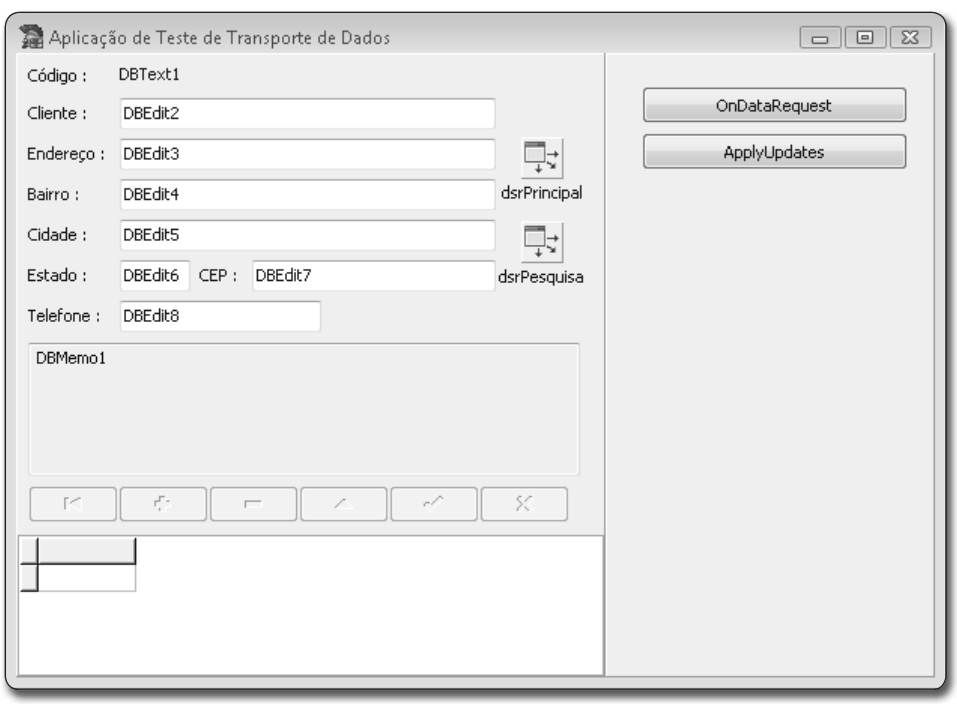


Figura 6. Interface do formulário principal da aplicação

Agora, implemente o evento *OnClick* do *btnRequest* conforme mostra a **Listagem 5.**

Na **Listagem 5** temos inicialmente a criação dinâmica do formulário de controle (**Linha 03**), em seguida é setado o parâmetro *OnDataRquest* na propriedade *TipoEvento* definida no formulário de controle. Na **Linha 07,** é verificado o estado do *ModalResult* do formulário. Sendo *mrOk* a variável privada *FDataSetPesquisa* será fechada, e como dito anteriormente o método *DataRequest* do *ClientDataSet* inicia uma chamada ao servidor de aplicação, informando os parâmetros antes da abertura. (**Linhas 09** à **11**). Caso o *ItemIndex* do *cmbTabela* seja 0, teremos o resultado da tabela de Clientes no nosso formulário de controle, neste caso, é passado como parâmetro ao *FDataSetPrincipal* que faz referência ao *cdsCliente*, o valor do campo chave da consulta, que será o campo *IDCLIENTE*. Por fim o *cdsCliente* é aberto retornando o registro referente aquela consulta, e o formulário é destruído e liberado da memória (**Linhas 12** à **22**).

Para finalizarmos, vamos a última codificação referente ao evento *OnClick* do *btnApply*, e que se encontra na **Listagem 6.**

Listagem 5. Evento OnClick do btnRequest

```
01  procedure TfrmAppCliente.btnRequestClick(Sender: TObject);
02  begin
03    frmControle := TfrmControle.Create(Self);
04    with frmControle do
05    try
06      TipoEvento := OnDataRequest;
07      if ShowModal = mrOk then
08      begin
09        FDataSetPesquisa.Close;
10        FDataSetPesquisa.DataRequest(
           VarArrayOf([
             cmbTabela.ItemIndex,
             cmbCampo.ItemIndex,
             edtConteudo.Text]));
11        FDataSetPesquisa.Open;
12        if cmbTabela.ItemIndex = 0 then
13        begin
14          FDataSetPrincipal.Close;
15          FDataSetPrincipal.FetchParams;
16          FDataSetPrincipal.Params[0].Value :=
17          FDataSetPesquisa.Fields[0].Value;
18          FDataSetPrincipal.Open;
19        end;
20      end;
21    finally
22      FreeAndNil(frmControle);
23    end;
24  end;
```

Listagem 6. Evento OnClick do btnApply

```
01  procedure TfrmAppCliente.btnApplyClick(Sender: TObject);
02  begin
03    if FDataSetPrincipal.ChangeCount > 0 then
04    begin
05      if Application.MessageBox('Deseja adicionar uma
          observação?','Confirmação!', MB_YESNO +
          MB_ICONQUESTION) = IDYES then
06      begin
07        frmControle := TfrmControle.Create(Self);
08        with frmControle do
09          try
10            TipoEvento := SetOptionalParam;
11            if ShowModal = mrOk then
12              FDataSetPrincipal.SetOptionalParam('PARAMETRO',
                  memParametros.Text ,True);
13          finally
14            FreeAndNil(frmControle);
15          end;
16      end;
17      FDataSetPrincipal.ApplyUpdates(0);
18    end;
19  end;
```

Nota do DevMan

O componente ClientDataSet possui dois estados em si:

ClientDataSet.Data: Refere-se ao conteúdo inicial, conteúdo as informações originais, sem qualquer alteração.

ClientDataSet.Delta: Refere-se ao DataPackage que sofreu alterações.

Comentando o código da **Listagem 6** podemos verificar que na **Linha 03** temos a verificação de alterações através do método *ChangeCount* do *ClientDataSet*. Em seguida é pedido ao usuário se ele deseja informar uma observação (**Linha 05**). Caso esta confirmação seja verdadeira, o formulário de controle novamente é criado dinamicamente, desta vez passando como parâmetro para a propriedade *TipoEvento* o valor *SetOptionalParam* (**Linha10**). Sendo *mrOk* o resultado do *ModalResult* do formulário, teremos o conteúdo informado no método *setOptionalParam* (**Linhas 11 e 12**). É importante comentar que o método *SetOptionalParam* é composto por três parâmetros. O primeiro consiste no nome do parâmetro adicional que será enviado ao servidor de aplicação. O segundo refere-se ao conteúdo que será enviado. E por último o terceiro parâmetro informa se este parâmetro estará presente no *Delta* do *ClientDataSet*. Finalmente, o formulário é destruído e liberado da memória e o *ApplyUpdates* é efetuado.

Com isso finalizamos nossa aplicação. Ao testá-la, verá que no momento em que se aplicam as alterações no banco de dados é exibida uma mensagem na tela que recebe a hora do servidor como parâmetro, para exibição.

Conclusão

Como pudemos ver neste artigo, existem várias formas de se realizar o tráfego de informações entre as camadas cliente e servidora. Estas práticas apresentadas, nos permitem uma maior facilidade na manipulação de dados através dos eventos do *DataSnap*. Utilizando-as corretamente e em conjunto é possível obter ótimos resultados de performance, e grande produtividade no seu dia-a-dia.

Uma boa opção para sempre adquirir mais informações específicas sobre a ocorrência e utilidade dos eventos, é desfrutar da própria ferramenta de *Help*. Como sempre, espero que todos tenham gostado do artigo. Um grande abraço e até a próxima! ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Existem coisas que não conseguimos ficar sem!

...só pra lembrar, sua assinatura pode estar acabando!

Renove Já!

www.devmedia.com.br/renovacao

DevMedia GROUP

Para mais informações:
www.devmedia.com.br/central

Nesta seção você encontra artigos sobre técnicas que poderão aumentar a qualidade do desenvolvimento de software

Introdução ao novo DataSnap 2009

Conheça as novidades desta tecnologia para desenvolvimento multicamadas

Adriano Santos
falecom@adrianosantos.pro.br
É desenvolvedor Delphi desde 1998, professor e programador PHP, bacharel em Comunicação Social e especialista em aplicações Win32. É Editor Chefe das revistas ClubeDelphi e WebMobile. Gerente de TI na SoftPark Informática e membro fundador do DUG-SP (www.dug-sp.com). Mantenedor do blog Delphi to Delphi (www.delphitodelphi.blogspot.com) com dicas, informações e tudo sobre o desenvolvimento Delphi.

Daniel Sobrinho Laporte
daniel.laporte@gmail.com
É editor técnico da revista ClubeDelphi. Graduado em Sistemas de informação e pós-graduado em administração de bancos de dados pela Universidade de Franca. É analista de sistemas e desenvolvedor Delphi desde 2005 desenvolvendo aplicações cliente/servidor e multicamadas em Delphi e Firebird.

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br
É natural de Araçatuba-SP e formado em Processamento de Dados pela FATEC-TQ. Atua como Analista Programador (Delphi e Java) há 3 anos, sendo 2 destes numa empresa do ramo de telecomunicações em minha cidade.

Resumo do DevMan

Aplicações multicamadas ou n-tier, como também são chamadas, estão sendo cada vez mais utilizadas no dia-a-dia por possibilitarem a separação das várias partes de um sistema em camadas distintas. Não só isso, mas o fato de se permitir a utilização de tecnologias win32 na web, de forma que se obtenham grandes requisitos de diferenciação como: alta velocidade, segurança, distribuição geográfica e uma série de outros fatores.

Apresentada como uma das novidades da nova versão do Delphi, o DataSnap 2009 já não exige o uso da tecnologia COM, da Microsoft, para conectividade remota entre aplicações. Essa característica o torna ainda mais fácil de ser utilizada por desenvolvedores de todos os níveis, do iniciante ao avançado.

Nesse artigo veremos

- Introdução ao DataSnap 2009;
- As mudanças em relação ao DataSnap tradicional;
- Facilidade em se construir uma aplicação multicamadas com esta nova tecnologia;

Qual a finalidade

Criar um aplicativo de demonstração simples da tecnologia DataSnap 2009. Este estará dividido em três camadas: Dados (BD), Negócios (Servidor de Aplicação) e Apresentação (Aplicações Cliente).

Quais situações utilizam esse recurso?

Para a construção das famosas aplicações multi-tier (multicamadas) com DataSnap, em que há uma separação do acesso ao banco de dados, a lógica de negócios e a interface do usuário.

Presente há muito tempo no *Delphi*, a tecnologia para desenvolvimento de aplicações multicamadas, conhecida anteriormente como *MIDAS* e mais tarde como *DataSnap*, ganhou uma novidade na versão mais recente da *IDE*. O *Delphi*, em sua recente versão 2009, apesar de ainda incluir o *DataSnap* clássico, oferece uma novidade baseada parcialmente na arquitetura *dbExpress*. Essa nova tecnologia está sendo chamada de *DataSnap 2009*. Nomenclatura essa que está sendo usada para evitar confusão com a *DataSnap* tradicional.

Uma série de novidades foram incluídas e até um *driver DataSnap* para facilitar o uso da tecnologia foi criado. Para entender melhor como tudo funciona,

veremos um artigo completo e a criação de um sistema para utilizar o máximo dos recursos disponíveis na nova tecnologia, portanto: mão na massa.

DataSnap versus DataSnap 2009

Em se tratando de mudanças e comparado a uma aplicação *tradicional DataSnap*, existem, no *DataSnap 2009*, algumas diferenças significativas. Mudanças essas que, no entanto, dizem respeito mais à arquitetura e à distribuição do que ao código em si. Vejamos algumas dessa diferenças:

Do lado servidor:

• No desenvolvimento do lado servidor, não há mais envolvimento com a tecnologia *COM*. Agora as aplicações (servidor e cliente) comunicam-se entre si diretamente sobre o protocolo *TCP/IP*;

• Sua implementação é um pouco mais complicada em termos de manipulação de componentes, porém há muito pouco código envolvido por trás se comparado à *COM*;

• Deve-se rodar manualmente a aplicação servidora, ou criar um serviço para ela;

• Não é mais necessário registrar o servidor nem qualquer outro serviço auxiliar. Tudo o que o servidor precisa é oferecer ao cliente uma porta *TCP/IP* aberta para acesso.

Do lado cliente:

• A implementação do lado cliente se mantém quase idêntico ao tradicional. Porém, para o *DataSnap 2009*, é necessário um componente *SQLConnection*, da paleta *dbExpress*, no lugar de qualquer objeto de conexão específico;

• Com o novo *driver DataSnap* do *dbExpress* (*driver* 100% *Delphi*), você não tem a necessidade de fazer o *deploy* de qualquer *DLL* no computador cliente, mesmo que na aplicação servidora esteja se fazendo uso também do *dbExpress* para a conectividade com o banco de dados.

Para termos uma idéia real de como funciona o desenvolvimento de um aplicativo com o novo *DataSnap*, iremos desenvolver uma aplicação de exemplo que conterá três camadas, conforme mostra a **Figura 1**.

Metodologia

O exemplo deste artigo utiliza a arquitetura cliente/servidor. Para mais informações sobre essa arquitetura e outras, visite *www.devmedia.com.br/articles/viewcomp.asp?comp=5219*

Figura 1. Arquitetura da aplicação de exemplo

Nota do DevMan

COM (Component Object Model) é uma tecnologia arquitetada pela Microsoft que viabiliza a comunicação entre aplicações client/server, realizada através de interfaces. Uma interface COM é a maneira como um objeto expõe sua funcionalidade ao meio externo.

Entendendo o exemplo

Faremos a criação de uma biblioteca digital, onde o usuário poderá consultar livros e até lê-los on-line. Claro que não implementaremos todas as funcionalidades da biblioteca por necessitar de um espaço maior na revista. Portanto, veremos os principais cadastros e as principais consultas. Construiremos um pequeno banco de dados para receber os cadastros e as consultas. Faremos isso utilizando o *Firebird 2.0* em conjunto com um *driver* de terceiro para acesso ao *BD* utilizando o *RAD Studio 2009*. Esse *driver* já é conhecido por muitos desenvolvedores *Delphi*, pois foi desenvolvido pelo colega *Thiago Borges*, de qualquer forma veremos como fazer a sua instalação e utilização na *IDE* do novo *Delphi*.

Criaremos então os seguintes cadastros no sistema:

• *Cadastro de títulos:* esse cadastro receberá as obras que estarão disponíveis no acervo, como livros, revistas em quadrinhos, enciclopédias e etc;

• *Cadastro de usuários/administradores:* os administradores do sistema serão os responsáveis por cadastrar os livros e alterar possíveis preferências de sistema. Consequentemente possibilitarão o acesso das consultas de acordo com seus níveis de acesso;

Em seguida faremos o desenvolvimento de uma aplicação servidora e uma aplicação cliente. O grande diferencial será o controle de permissões. Os administradores possuirão a permissão para o acesso do controle dos cadastros de títulos, juntamente com o cadastramento de novos usuários/administradores. Já os usuários comuns, não possuirão permissão de acesso aos cadastros, e sim apenas as consultas de obras e outros usuários cadastrados que não sejam administradores do sistema.

Criando o banco de dados

Como nossa aplicação não será complexa ao extremo, vamos criar uma base bem simples para que entendamos como funcionará nossa aplicação. A idéia não é criar um exemplo muito extenso e sim entender como criar um sistema na nova tecnologia. Isso nos dará base para construir qualquer tipo de software usando o *DataSnap 2009*.

Para iniciar o projeto, vamos primeiramente criar o banco de dados e as tabelas necessárias para cada tela de cadastro. As tabelas são: *Titulos* e *Usuarios*. A tabela de usuários terá um campo que indicará se ele é ou não um administrador. Neste artigo, criaremos o banco de dados utilizando a ferramenta *IBExpert* em sua versão *Standard*. Por isso, acesse o link *www.ibexpert.com* e em seguida entre no item *IBExpert*.

Nota

<CaminhoBanco>\BIBLIOTECA-DIGITAL.FDB pelo diretório do banco de dados e seu nome, ex: C:\BancoDeDados\Biblioteca-Digital.fdb.

À esquerda do site do fabricante clique em *Download>Free*. Preencha o formulário de cadastro e aguarde o e-mail com as instruções de download da ferramenta. Após isso, abra o *IBExpert* e vamos criar o banco de dados usando a opção *Script Executive* presente no menu *Tools*.

Com a tela de *scripts* aberta, digite o código da **Listagem 1**. Nele estão contidos os esquemas para a criação do banco de dados assim como de cada tabela do sistema. Digitado o *script*, basta executá-lo usando o botão *Run Script* ou pressione a tecla F9.

Iniciando o exemplo

Muito bem, já criamos nosso banco de dados e o preparamos para receber informações. O próximo passo é preparar um ambiente de desenvolvimento onde faremos a instalação do *RAD Studio 2009* e então iniciaremos a codificação do sistema. Rapidamente vamos ver como efetuar o download e instalação do *Delphi 2009*.

Acesse o site da empresa *Embarcadero Technologies* em *www.embarcadero.com* e em seguida passe o mouse sobre *Products*. Localize *RAD Studio* e clique nesse *link*. Veja que o *banner* central se modifica (**Figura 2**), então basta clicar em *Download Trial*. Na tela seguinte encontraremos vários links, basta clicar no link que melhor te atenda. Um e-mail com a chave de liberação *Trial* será enviada ao seu e-mail.

ClubeDelphi PLUS www.devmedia.com.br/clubedelphi/portal.asp

Acesse agora o mesmo o portal do assinante ClubeDelphi e assista a uma vídeo aula de Luciano Pimenta que mostra como baixar e instalar o IBExpert em sua versão gratuita.

www.devmedia.com.br/articles/viewcomp.asp?comp=3082

Se preferir acesse diretamente o *link http://downloads.embarcadero.com/free/delphi* e faça o *download*. É necessário que se tenha um usuário e senha no site. Para quem já tinha cadastro no portal da *CodeGear*, o mesmo valerá para *download* desse *Trial*.

Instalando o driver de conexão

Como mencionado no início do artigo, faremos uso do *driver* do colega *Thiago Borges* para conexão com o *Firebird* através do *DBX4*, *dbExpress 4*. O *driver* pode ser encontrado no site do autor em *http://www.tbosystems.bluehosting.com.br/dbx4/*, bem como as informações necessárias para instalação.

Basicamente, após ter realizado o *download*, iremos proceder com a instalação dele através do auto-instalador. Execute-o, e na primeira tela clique no botão *Next*. Na segunda tela selecione o diretório de instalação de sua preferência e em seguida avance novamente. Por último clique em *Install* para que o processo seja concluído. Se tudo ocorrer bem o seu resultado será o mesmo da **Figura 3.**

Agora que finalizamos a instalação do *driver* a ser utilizado na aplicação, passaremos para o processo de modelagem e criação do servidor.

Listagem 1. Script de criação do banco

```
CREATE DATABASE '<CaminhoBanco>Biblioteca-Digital.fdb'
USER 'SYSDBA' PASSWORD 'masterkey'

CREATE GENERATOR GEN_TITULOS_ID;
SET GENERATOR GEN_TITULOS_ID TO 3;

CREATE GENERATOR GEN_USUARIOS_ID;
SET GENERATOR GEN_USUARIOS_ID TO 5;

CREATE TABLE TITULOS (
    ID              INTEGER NOT NULL,
    TITULO          VARCHAR(100),
    AUTOR           VARCHAR(100),
    ISBN            VARCHAR(30),
    EDITORA         VARCHAR(60),
    VALOR           DOUBLE PRECISION,
    QTDE_PAGINAS    INTEGER,
    ANO_PUBLICACAO  INTEGER,
    STATUS          VARCHAR(30)
);

CREATE TABLE USUARIOS (
    ID             INTEGER NOT NULL,
    NOME           VARCHAR(100),
    APELIDO        VARCHAR(60),
    SENHA          VARCHAR(10),
    ADMINISTRADOR  CHAR(1)
);

INSERT INTO TITULOS (ID, TITULO, AUTOR, ISBN, EDITORA, VALOR,
QTDE_PAGINAS, ANO_PUBLICACAO, STATUS) VALUES (1, 'Dom Casmurro',
'Machado de Assis', '178203008300', 'Ática', 55, 120, 1973,
'Emprestado');
INSERT INTO TITULOS (ID, TITULO, AUTOR, ISBN, EDITORA, VALOR,
QTDE_PAGINAS, ANO_PUBLICACAO, STATUS) VALUES (2, 'A ilha perdida',
'Alfredo Munhoz', '78991838773', 'Editora Globo', 12, 34, 1988,
'Emprestado');
INSERT INTO TITULOS (ID, TITULO, AUTOR, ISBN, EDITORA, VALOR, QTDE_
PAGINAS, ANO_PUBLICACAO, STATUS) VALUES (3, 'O escaravelho do diabo',
'Mário Borges', '890109880', 'Ática', 15, 130, 1999, 'Alugado');

COMMIT WORK;

INSERT INTO USUARIOS (ID, NOME, APELIDO, SENHA, ADMINISTRADOR)
VALUES (2, 'Adriano Santos', 'asantos', '123456', 'S');
INSERT INTO USUARIOS (ID, NOME, APELIDO, SENHA, ADMINISTRADOR)
VALUES (3, 'Daniel Sobrinho', 'dsobrinho', '334333', 'S');
INSERT INTO USUARIOS (ID, NOME, APELIDO, SENHA, ADMINISTRADOR)
VALUES (4, 'Guinther Pauli', 'gpauli', '889900', 'N');
INSERT INTO USUARIOS (ID, NOME, APELIDO, SENHA, ADMINISTRADOR)
VALUES (5, 'Gladstone Matos', 'gmatos', '8t88s00', 'N');

COMMIT WORK;

ALTER TABLE TITULOS ADD CONSTRAINT PK_TITULOS PRIMARY KEY (ID);
ALTER TABLE USUARIOS ADD CONSTRAINT PK_USUARIOS PRIMARY KEY (ID);

SET TERM ^ ;

CREATE TRIGGER TITULOS_BI FOR TITULOS
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  IF (NEW.ID IS NULL) THEN
    NEW.ID = GEN_ID(GEN_TITULOS_ID,1);
END
^

CREATE TRIGGER USUARIOS_BI FOR USUARIOS
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  IF (NEW.ID IS NULL) THEN
    NEW.ID = GEN_ID(GEN_USUARIOS_ID,1);
END
^

SET TERM ; ^
```

Figura 2. Banner de download do RAD Studio 2009

Figura 3. Finalização da instalação do driver

Criando o servidor

Primeiro construiremos o servidor de aplicação, dessa forma fica mais simples construir as aplicações cliente depois. O *Server Application*, como pode ser chamado, será uma aplicação *VCL (Visual Component Library)* comum. Para isso, com o *Delphi 2009* aberto, inicie um novo projeto usando *File>New>VCL Forms Application – Delphi*. Salve o projeto como *DataSnapServer2009.dprj* e o formulário *(frmDataSnapServer)* como *untDataSnapServer.pas*. De preferência crie uma pasta para armazenar seu projeto e dentro dela, crie outras pastas para cada projeto que criaremos.

Antes de prosseguirmos, iremos alterar alguns detalhes no nosso formulário. Primeiramente adicione três *Label's (lblData, lblHora, lblDescricao)*. Altere o *Caption* do *lblDescricao* para *DataSnap Server 2009* e insira um componente *Timer* e um *TrayIcon*. Ative a propriedade *Visible* do *TrayIcon* para *True*, e no evento *OnDblClick* adicione o seguinte código:

```
Self.Visible := not Self.Visible;
```

Selecione o *Timer* e no evento *OnTimer* o código a seguir:

```
lblData.Caption := 'Data : ' + FormatDateTime('dd/mm/yyyy', Date);
lblHora.Caption := 'Hora : ' + FormatDateTime('hh:nn:ss', Now);
```

Por último, no evento *OnCreate* do formulário faça a seguinte implementação:

```
Application.ShowMainForm := False;
```

Nota

A criação de uma nova conexão com o banco de dados que será criado, será detalhada mais adiante, porém é altamente recomendável que visite a página do autor disponível no mesmo link de download mencionado anteriormente para assim tomarem nota dos demais requisitos, relativos à distribuição por exemplo.

O recomendado é que se crie um *Data Module*, que é tratado como um *Container* de objetos do servidor de aplicação. Deste modo, crie um novo *Data Module* acessando o menu *File>New>Other>Delhi Files>Data Module*. Iremos chamá-lo de *DMServerContainer* e sua *Unit* será denominada *untDMServerContainer.pas*.

O próximo passo será inserir um componente *SQLConnection (sqlConexao)*, da paleta *dbExpress*, para conexão ao banco de dados e configurá-lo.

Na versão do *Delphi 2009*, diferentemente das anteriores, a conexão no *SQLConnection* não é configurada através de um duplo clique no próprio componente , mas sim através do *Data Explorer* que encontra-se junto ao *Project Manager (View>Project Manager* ou *Ctrl + Alt + F11*) e que após a instalação se encontrará do mesmo modo da **Figura 4.**

Figura 4. Janela do Data Explorer

Clique com o botão direito sobre o *TDBODBXFB* e selecione a opção *Add New Connection*. Na janela a seguir informe o *Connection Name Biblioteca-Digital* e clique em *OK*. Expandindo sua ramificação, perceba que foi adicionada uma conexão, porém sem configuração alguma. Clique sobre a conexão e com o botão direito selecione a opção *Modify Connection* e em seguida clique no botão *Advanced*. Será exibida uma janela para a configuração (**Figura 5**) do banco de dados, configure-a conforme mostra a **Figura 6**. Lembrando que o *Password* do banco de dados é *masterkey*.

Após ter finalizado esta etapa, selecione o componente *sqlConexao* e modifique as propriedades *Driver* para *TBODBXFB*, *ConnectionName* para *Biblioteca-Digital*, *LoginPrompt* para *False* e *VendorLib* para *fbclient.dll*. Seguindo estes dados modifique a propriedade *Connected* para *True*, e perceberá que o componente já consegue se conectar ao banco de dados com êxito. Agora

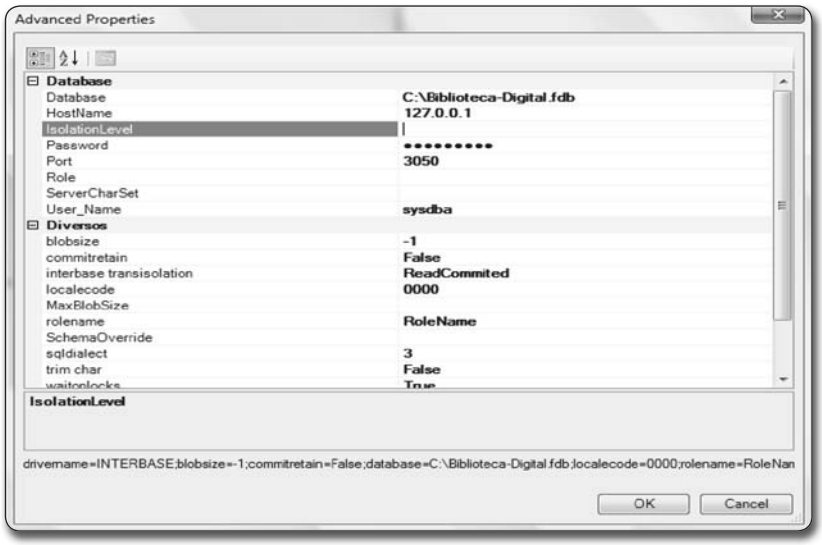


Figura 5. Propriedades avançadas do Data Explorer

que concluímos a conexão com o banco vamos configurar os componentes do *Data Module.*

As novidades relativas ao novo *DataSnap* , ficarão por conta dos próximos passos descritos. Primeiramente, faremos a inclusão de três novos componentes ao *Container* do nosso projeto. Eles serão os responsáveis por toda a parte de configuração e conectividade do nosso servidor de aplicação, em substituição ao suporte *COM*. Insira então, os seguintes componentes da paleta *DataSnap Server.*

Adicione um *DSServer (Server)*, um *DSServerClass (SCPrincipal)* e um *DSTCPServerTransport (ServerTransport)*. Ao final deste passo, o seu *DMServerContainer* deverá estar conforme a **Figura 7**.

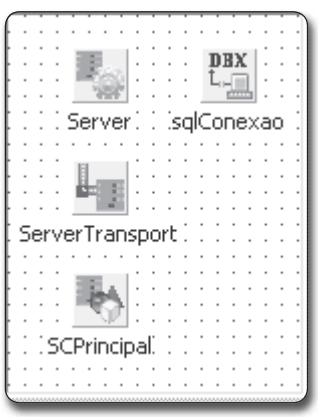


Figura 6. Container de objetos da aplicação servidora

Agora, para um melhor entendimento, iremos configurar estes três componentes, ao mesmo tempo em que é apresentada uma breve descrição de cada um.

DSServer:

É o componente principal da configuração do servidor. Faz a junção de todos os outros componentes restantes do *DataSnap 2009*. Neste exemplo, manteremos sua configuração padrão.

DSTCPServerTransport:

É quem define o protocolo de transporte a ser usado e suas configurações. Um exemplo é a porta *TCP/IP* (único protocolo direto disponível no *Delphi 2009*) a ser usada para comunicação das aplicações (cliente e servidor). Para o nosso caso, faça apenas a ligação de sua propriedade Server com nosso componente *Server*, deixando o restante com os valores padrões.

DSServerClass:

É o componente necessário para cada classe que você queira expor (não é a classe em si, mas sim, uma fábrica de objetos da classe que será chamada por um cliente remoto). Para este exemplo, altere suas propriedades *Server* para *Server* (nosso componente) e *LifeCyle* para *Session*. Uma atenção especial deve ser dada para esta última, uma vez que, o ciclo de vida dos objetos no *DataSnap 2009* dependem do valor desta. *LifeCycle* pode assumir três valores (três constantes da classe *TDSLifeCycle*, definida na *Unit DSNames*):

• *Session:* indica que o servidor criará um objeto diferente para cada conexão cliente (*socket connection*), isto é, um objeto do servidor para cada cliente. Esses objetos são liberados quando a conexão é fechada;

• *Invocation:* indica que um novo objeto do servidor será criado (e destruído) toda vez que o método deste for invocado;

• *Server:* utiliza a metodologia do padrão *Singleton* trabalhando com um objeto do servidor compartilhado. Cada cliente usa a mesma instância, inclusive os mesmos dados.

Antes de concluirmos as configurações de nosso Container, é necessário criarmos nosso *Server Module*. Siga pelo menu *File>New>Other>Delphi Project>Delphi Files>Server Module*. Se tiver dúvidas consulte a **Figura 7**. Salve-o como *untSMPrincipal.pas* e mude sua propriedade *Name* para *SMPrincipal*.

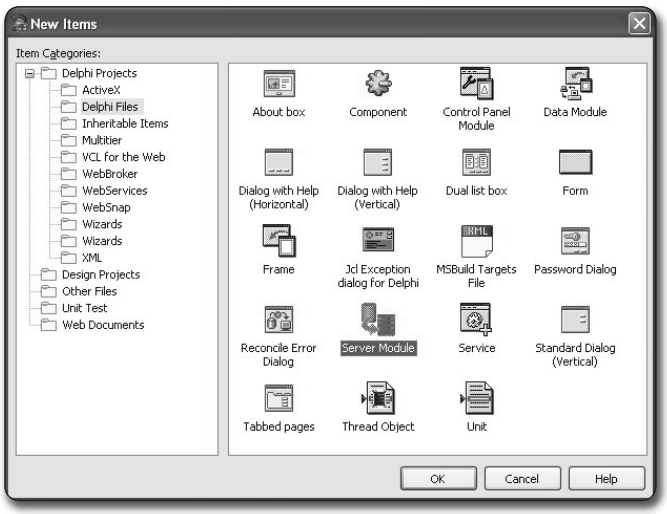


Figura 7. Localização do item Server Module

A classe *TDSServerModule* herda de *TDataModule*, incluindo a interface *IAppServer* (a mesma interface usada pela tecnologia *COM*, através do *TRemoteDataModule*). Ela já traz a diretiva de compilação *$MethodInfo* habilitada, que é um dos requisitos necessários para invocação remota de métodos no *DataSnap 2009*.

Adicione no seu novo *Server Module* três *SQLDataSet's (sqlTitulos, sqlUsuarios, sqlPesquisa)*, e em seguida três *DataSetProvider's (dspTitulos, dspUsuarios, dspPesquisa)*. Inclua à cláusula *uses* as *Unit's DBClient* e *Variants*, e por último, adicione a *untDMServerContainer*. Ligue os *SQLDataSet's* no *sqlConexao* (localizado no *DMServerContainer*) através da propriedade *SQLConnection*. Em seguida adicione as sentenças *SQL* aos respectivos componentes conforme mostra a **Listagem 2.**

Adicionadas as sentenças, configure os parâmetros através da propriedade *Params*. Dê um duplo clique sobre os componentes e adicione os *Provider Flags* no *Fields Editor*. Configure os campos *ID* como [*pfInKey*, *pfInUpdate* e *pfInWhere*], e nos demais deixe habilitada somente a opção [*pfInUpdate*]. Para

Listagem 2. Sentenças SQL dos componentes SQLDataSet

```
[sqlTitulos]
SELECT
  TITULOS.ID,
  TITULOS.TITULO,
  TITULOS.AUTOR,
  TITULOS.ISBN,
  TITULOS.EDITORA,
  TITULOS.VALOR,
  TITULOS.QTDE_PAGINAS,
  TITULOS.ANO_PUBLICACAO,
  TITULOS.STATUS
FROM
  TITULOS
WHERE

  TITULOS.ID =:ID

[sqlUsuarios]
SELECT
  USUARIOS.ID,
  USUARIOS.NOME,
  USUARIOS.APELIDO,
  USUARIOS.SENHA,
  USUARIOS.ADMINISTRADOR
FROM
  USUARIOS
WHERE USUARIOS.ID =:ID
```

Nota

Devido a extensão da nossa aplicação, algumas partes do código não serão comentadas, como o método de pesquisa por exemplo, porém, o projeto completo se encontra disponível para download no site da DevMedia.

os *DataSetProvider's* configure o *dspTitulos* e o *dspUsuarios* com o *UpdateMode upWhereKeyOnly* e em *Options* marque [*poIncFieldProps*, *poPropagateChanges*]. Para o *dspPesquisa* apenas modifique *Options*, configurando o [*poAllowCommandText*].

Configurações feitas, hora de partirmos para a codificação. Criaremos duas rotinas na sessão *public* do nosso *Server Module* a primeira será responsável por validar nosso usuário no ato do *Login* do sistema, permitindo ou não a autenticação do mesmo. A segunda tratará de retornar as consultas à aplicação cliente. Acompanhe o código na **Listagem 3.**

Listagem 3. Função de Login do sistema

```
01  function TSMPrincipal.Login(Usuario, Senha: String; var Adm :
     Boolean): Boolean;
02  begin
03    sqlPesquisa.Close;
04    sqlPesquisa.CommandText :=
        Format(
        ' SELECT ' + #13 +
        ' USUARIOS.ADMINISTRADOR ' + #13 +
        ' FROM USUARIOS WHERE ' + #13 +
        ' USUARIOS.APELIDO = %s AND ' + #13 +
        ' USUARIOS.SENHA   = %s ', [QuotedStr(Usuario),
           QuotedStr(Senha)]);
05    sqlPesquisa.Open;
06    Adm := sqlPesquisa.Fields[0].AsString = 'S' ;
07    Result := (not sqlPesquisa.Fields[0].IsNull);
08  end;
```

Acompanhando o código da **Listagem 3**, podemos observarue para consulta é utilizado o *sqlPesquisa*. Este, inicialmente tem atribuído a instrução que será utilizada para validar se o usuário é um administrador, passando como parâmetros o usuário e senha recebidos através da aplicação cliente (**Linha 04**). Perceba que temos uma variável de tipo *Boolean* denominada de *Adm* (**Linha 06**). Esta será uma variável vinda da aplicação cliente, e atribuirá o valor *True* caso o campo *ADMINISTRADOR* esteja preenchido com o valor *S*, em qualquer ocasião diferente o valor passado para esta variável será *False*. A rotina retornará um valor *True* para qualquer valor obtido diferente de nulo. (**Linha 07**).

Para completar, vamos abrir o *DMServerContainer* e adicioná-lo a *Unit* deste novo *Server Module* criado, e em seguida codificar evento *OnGetClass* do componente *SCPrincipal*. Este irá retornar a classe do servidor que queremos expor para outras aplicações, ou seja, todos os métodos públicos desta classe que foram criados, poderão ser chamados por uma aplicação cliente. No nosso caso, tal classe será nosso próprio *Server Module* (*TSMPrincipal*). Codifique o evento conforme o código a seguir:

```
PersistentClass:= TSMPrincipal;
```

Feito isso, aperte *F9* para rodar a aplicação. Deixe-a rodando, uma vez que, ela será nosso servidor de aplicações e deverá estar ativo para que a aplicação cliente funcione corretamente.

Criando a primeira aplicação cliente

Com o nosso servidor de aplicações construído e já rodando, iniciaremos agora o desenvolvimento da nossa aplicação cliente que será do tipo Win32. Ainda com a *IDE* do *Delphi* aberta, adicione um novo projeto *VCL* ao nosso grupo de aplicações, utilizando o *Project Manager*.

Salve o projeto como *Administracao.dpr*, o formulário como *untPrincipal.pas* e o grupo de projetos como *DataSnapGroup.groupproj*. Com nosso novo projeto aberto, adicione um *DataModule*, nomeando-o como *DMPrincipal* e salve-o como *untDMPrincipal.pas*. Nele, adicione um *SQLConnection (ConexaoClient)*, com sua propriedade *Driver* apontada para o novo *driver DataSnap* e *LoginPrompt* como *False*. Além disso, ele deve estar configurado para a porta *TCP/IP* disponibilizada pelo nosso servidor de aplicações (configurada no componente *ServerTransport*). Como deixamos definido o valor como *211*, que é o valor padrão para a porta, não precisaremos alterá-la aqui também.

Adicione agora dois *DSProviderConnection's* (*DSPCUsuarios*, *DSPCTitulos*) que serão usados para fazer referência à classe que será persistida pelo servidor de aplicação, através de sua propriedade *ServerClassName*. Portanto, em sua propriedade *SQLConnection*, aponte para o nosso componente *ConexaoClient* e em *ServerClassName* digite *TSMPrincipal*, que nada mais é que o nome da classe que expomos no servidor e que queremos acessar pelo cliente.

Continuando, adicione dois *ClientDataSet's* (*cdsUsuarios*, *cdsTitulos*). Assim como em uma aplicação *DataSnap* tradicional, ele poderá buscar e atualizar um *DataSet* remoto. Para configurá-los, ligue-os ao através de suas propriedades *RemoteServer* aos respectivos *DSProviderConnection's*. Se você obteve êxito ao ativar o *ConexaoClient*, ao fazer isto, perceberá que em *ProviderName* já é carregada a lista de *DataSetProvider's* disponíveis no nosso servidor de aplicação. Novamente configure-os vinculando-os aos seus nomes correspondentes.

Na sessão *public* do *Data Module*, crie uma nova variável de tipo *Boolean*, conforme o seguinte código:

```
private
  { Private declarations }
public
  FAdm : Boolean;
  { Public declarations }
end;
```

Por último adicione o código da listagem a seguir aos eventos *AfterPost* dos dois *ClientDataSet's*:

```
TClientDataSet(DataSet).ApplyUpdates(0);
```

Com nosso *Data Module* pronto, abra agora o formulário criado junto do projeto, *untPrincipal*. Nele, modifique as propriedades *Name* para *frmPrincipal*, *Caption* para *Biblioteca Digital*, *Position* para *poDesktopCenter*, *Width* para *800* e por último *Height* para *600*. Adicione em seguida um *MainMenu* e um *ActionList* (*Acoes*). No *ActionList*, dê um duplo clique e com o botão direito do mouse e adicione duas ações (*New Action*), chamadas de *actUsuarios* e *actTitulos*. Altere seus *Caption's* respectivamente para *Usuários* e *Títulos*. Altere agora o *MainMenu* inserindo dois itens de menu

Nota do DevMan

Singleton é um padrão de projeto de software (do inglês Design Pattern). Este padrão garante a existência de apenas uma instância de uma classe, mantendo um ponto global de acesso ao seu objeto. Uma forma bastante adotada é a utilização de uma variável global para este modelo, porém, este método muitas vezes não garante esta única instância. Um exemplo claro de sua aplicação pode ser implementado em log de dados, onde é possível criar uma instância de log para a aplicação inteira.

(*Consulta* e *Cadastro*), e mais dois sub-itens de menu em cada um dos menus. Ligue-os nas ações criadas no *ActionList*, de forma que ambos itens do menu fiquem com os sub-itens *Usuários* e *Títulos*. Após concluir o *Layout* seu *frmPrincipal* deverá estar com a aparência semelhante ao da **Figura 8.**

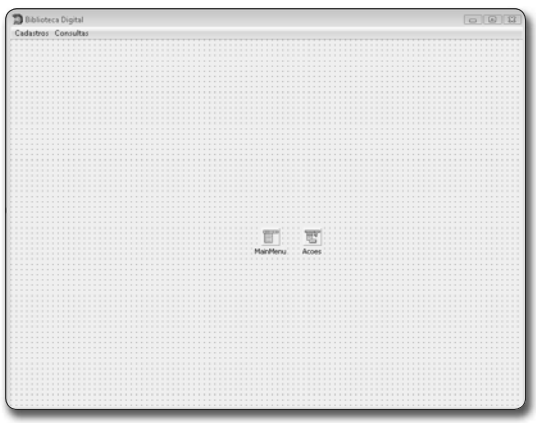


Figura 8. Aparência do formulário principal da aplicação cliente Win32

Feito isso, por enquanto manteremos este formulário como está. O próximo passo que iremos implementar será a criação da nossa tela de *Login* que será encarregada de validar nossos usuários já cadastrados no banco de dados. Deste modo siga no menu *File>New>Form – Delphi*, para que o nosso formulário seja carregado. No novo *Form* altere as propriedades *Name* para *frmLogin*, *Position* para *poMainFormCenter*, *BorderStyle* para *bsDialog*, *Caption* para *Login*, *Width* para *216* e *Height* para *250*. Insira um *Panel* (*pnlTopo*), três *Label's (Usuário, Senha, Autenticação de Usuários)*, dois *Edit's (edtUsuario, edtSenha)* e dois *BitBtn's (btnLogar, btnSair)*. Limpe as propriedades *Text* dos *Edit's* e modifique a propriedade *ModalResult* do *btnSair* para *mrCancel*. Veja como ficou nossa tela de *Login* na **Figura 9**.

Antes que possamos prosseguir, voltaremos a atenção ao nosso *DMPrincipal*. Neste momento, temos a necessidade de acessar nossa aplicação servidora para obter o método de *Login*, anteriormente criado. Para isso vamos conhecer mais alguns recursos do novo *DataSnap*.

Chamada de métodos do servidor

Assim como antes, o novo *DataSnap* permite que você escreva métodos no servidor que poderão ser chamados por

Figura 9. Formulário de Login

uma aplicação cliente. A novidade é que agora as chamadas remotas destes métodos são baseadas na *RTTI* (*Runtime Type Information*) do *Delphi*. Outra novidade é que a passagem de parâmetros para estes métodos, quando necessária, é baseada nos tipos dos parâmetros do *dbExpress*.

Para fazermos a chamada a esse método que implementamos, existem basicamente duas maneiras. A primeira seria utilizar o novo componente *SQLServerMethod* da paleta *dbExpress*, fazendo a chamada ao método do servidor como se fosse uma *stored procedure* (inclusive podendo fazer uso de seus parâmetros de entrada e saída). A segunda seria gerar uma classe *Proxy* na aplicação cliente e utilizar a mesma para fazer a chamada.

Para nosso exemplo iremos abordar a criação da classe *Proxy*. Portanto abra o seu *Data Module*, e com o servidor de aplicação em execução clique sobre o componente *ConexaoClient* com o botão direito do mouse e selecione a opção *Generate DataSnap client classes*, conforme mostra a **Figura 10.**

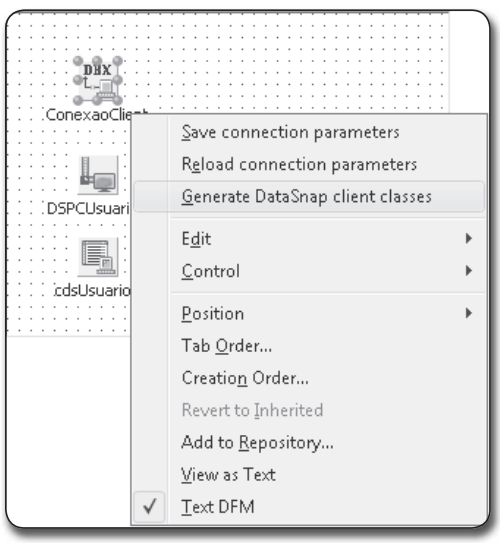
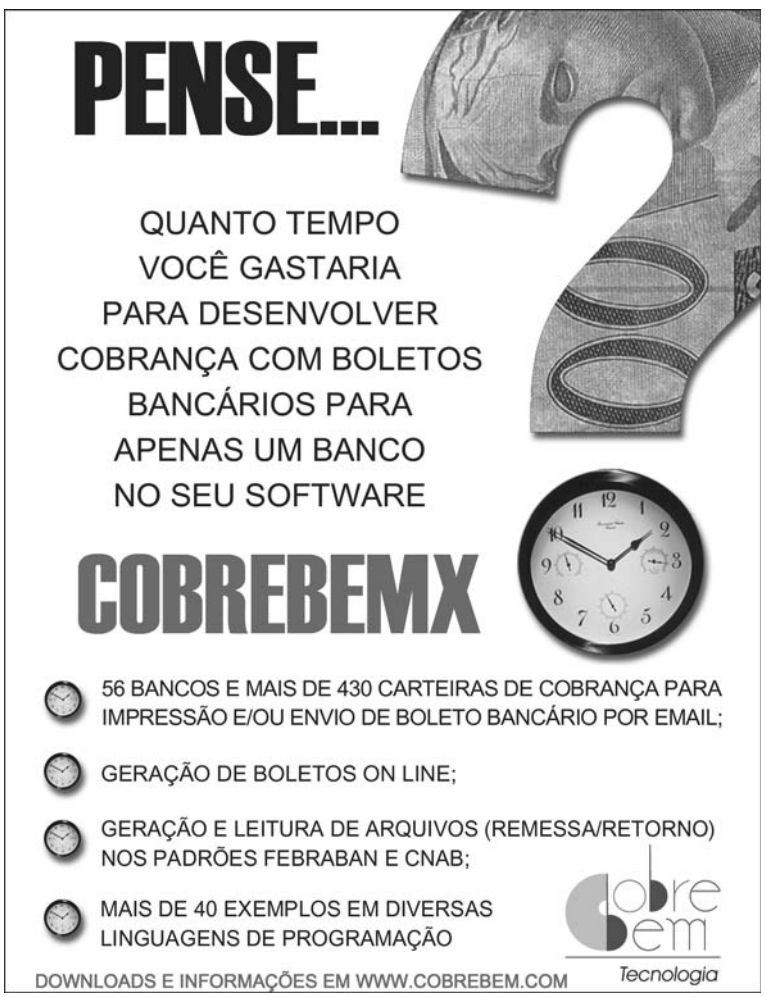


Figura 10. Gerando a Proxy Client

Veja que o Delphi nos gerou automaticamente uma classe que podemos usar para chamar os métodos do servidor. Em outras palavras, ela fará o mesmo processo do componente *SQLServerMethod*, via código, ao invés do uso do componente. Salve o seu projeto e nomeie o formulário anteriormente criado como *untLogin.pas* e a nova *Unit* de classes como *untProxyClient.pas*. Volte ao *frmLogin*, e adicione à *uses* as *Unit's untProxyClient* e *untDMPrincipal*, em seguida faça a codificação do evento OnClick do botão *btnLogar*, conforme ilustra a **Listagem 4.**

Listagem 4. Chamando método pela classe Proxy

```
01  procedure TfrmLogin.btnLogarClick(Sender: TObject);
02  var
03    Login : TSMPrincipalClient;
04  begin
05    try
06      try
07        DMPrincipal.ConexaoClient.Connected := True;
08        Login := TSMPrincipalClient.Create(DMPrincipal.
           ConexaoClient.DBXConnection);
09        if Login.Login(edtUsuario.Text, edtSenha.Text,
           DMPrincipal.FAdm) then
10          Self.ModalResult := mrOk
11        else
12          ShowMessage('Usuário ou senha inválidos !');
13      except
14        DMPrincipal.ConexaoClient.Connected := False;
15        ShowMessage('Problemas ao conectar');
16      end;
17    finally
18      FreeAndNil(Login);
19    end;
20  end;
```



Observando o código da **Listagem 4,** podemos perceber que na **Linha 03** é criada uma variável que referencia a classe de persistência definida no servidor de aplicação, ou seja, nosso *Server Module.* Todo o trecho de código foi isolado de erros através das instruções de captura de exceção. Na **Linha 07** a conexão remota é tentada. Na próxima linha (**Linha 08**), o objeto que foi declarado como um tipo referente à nossa classe é criado e instanciado, tendo como passagem de parâmetro a conexão em si, através da propriedade *DBXConnection.* Feito isso, a nossa rotina é executada, passando todos os parâmetros necessários. Atente para a variável pública anteriormente declarada no *Data Module* que é passada e atribuída no servidor de aplicação (**Linha 09**). Esta variável será responsável pelo controle de visibilidade dos nossos formulários de acordo com as restrições impostas pelos administradores do sistema. Caso esta rotina nossa retorne *True*, a autenticação será permitida através do *mrOk* retornado pelo *ModalResult,* caso contrário a mensagem de erro é mostrada e o formulário continua em exibição (**Linhas 09** a **12**). Em caso de erros referentes a conexão, esta automaticamente será encerrada, e a mensagem de problemas será exibida (**Linhas 14** e **15**). Finalizando, temos a destruição da variável exibida na **Linha 18.**

Passaremos agora para a criação do nosso formulário de herança que será único, tanto para o módulo de consulta quanto para o método de cadastro. Portanto, novamente acesse o menu *File>New>Form – Delphi.* Renomei-o para *frmHeranca* e altere sua *Position* para *poMainFormCenter.* Insira à *uses* do seu novo *Form* as *Unit's DBClient, untProxyClient* e *untDMPrincipal.* Adicione um *PageControl* (*pclPrincipal*) e duas abas (*tbsCadastro, tbsConsulta*), seguidas de dois *DataSource's* (*dsrPesquisa, dsrCadastro*). Na *tbsCadastro,* inclua um *DBNavigator* e ligue-o ao *dsrCadastro* por meio da propriedade *DataSource.* Na aba de consulta insira um *Label* (*Pesquisa*), um *ComboBox* (*cmbPesquisa*), um *Edit* (*edtPesquisar*), um *BitBtn* (*btnPesquisar*) e uma *DBGrid* ligando-a ao *dsrPesquisa* também através da propriedade *DataSource*. Em seguida vamos partir para a codificação.

Declare uma variável *private* no seu formulário chamada *FDataSetPesquisa* como mostrado a seguir:

```
FDataSetPesquisa : TClientDataSet;
```

Crie três propriedades no escopo *public* do seu *Form* como mostrado a seguir e pressione simultaneamente as teclas *Ctrl + Shift + C* para que os métodos sejam declarados automaticamente:

```
property DataSetCadastro : TClientDataSet;
property TipoTela : Integer;
property Administrador : Boolean;
```

Seguindo a ordem dos métodos *Set* que foram implementados, iremos modificar algumas ações, primeiramente no corpo do método *SetDataSetCadastro*:

```
FDataSetCadastro := Value;
FDataSetCadastro.Open;
dsrCadastro.DataSet := FDataSetCadastro;
```

Nota do DevMan

RTTI (uma abreviação de Runtime Type Information, que em português tem o significado de Informação de Tipo de Dado em Tempo de Execução) pode ser definida como uma técnica disponível em algumas linguagens de programação que consiste em manter informação sobre o tipo de dado de um objeto em memória enquanto um programa encontra-se em execução no computador.

Nota do DevMan

Deve-se prestar muita atenção ao se fechar uma aplicação servidora DataSnap 2009, pois, ao contrário da arquitetura COM, ela não nos adverte sobre conexões pendentes à ela. Em casos como este, o servidor seria visualmente fechado, porém, permaneceria em funcionamento na memória do computador. Para ser finalizado com sucesso seria necessário utilizar o gerenciador de tarefas do Windows. Outro caso comum é o da própria IDE do Delphi automaticamente abrir uma conexão ao Server para consultar suas classes e métodos disponíveis e, conseqüentemente, esta conexão segurar o servidor, quando este for fechado.

Listagem 5. Método de pesquisa do formulário de Herança

```
01  procedure TfrmHeranca.btnPesquisarClick(Sender: TObject);
02  var Pesquisa : TSMPrincipalClient;
03  begin
04    try
05      Pesquisa :=
          TSMPrincipalClient.Create(DMPrincipal.ConexaoClient.
           DBXConnection);
06      FDataSetPesquisa.Close;
07      FDataSetPesquisa.Data :=
          Pesquisa.Pesquisa(VarArrayOf([
          FTipoTela,
          cmbPesquisa.ItemIndex,
          edtPesquisar.Text,
          FAdministrador]));
08    finally
09      FreeAndNil(Pesquisa);
10    end;
11  end;
```

Em seguida no método *SetAdministrador,* que ocultará as abas conforme o nível de administração que será recebido na propriedade, conforme exibe o código a seguir:

```
FAdministrador := Value;
tbsCadastro.TabVisible := FAdministrador;

tbsConsulta.TabVisible := not FAdministrador;
```

No evento *OnCreate* do seu *frmHeranca* adicione o seguinte código:

```
if not Assigned(FDataSetPesquisa) then
  FDataSetPesquisa := TClientDataSet.Create(Self);

dsrPesquisa.DataSet := FDataSetPesquisa;
```

O código mostrado anteriormente, fará com que um *ClientDataSet* dinâmico seja criado para a consulta no momento da criação do formulário. Logo, necessitamos fazer a liberação do mesmo no momento em que o formulário é liberado da

memória, portanto, implemente o evento *OnDestroy* conforme a listagem que se segue:

```
FreeAndNil(FDataSetPesquisa);
```

Para finalizar, o nosso formulário criaremos o método genérico de consulta, que é exibido detalhadamente na **Listagem 5.**

Na **Listagem 5** temos novamente a referência a um método criado em nosso servidor de aplicação. Acompanhando, temos na **Linha 02** a declaração da variável da classe servidora. Em seguida o método é instanciado novamente com o parâmetro da conexão (**Linha 05**). Nas **Linhas 06** e **07**, o *ClientDataSet* de pesquisa criado em tempo de execução é fechado e em seguida recebe o resultado do método *Pesquisa* que retorna um conjunto *OleVariant*. Repare que os parâmetros são passados através do método *VarArrayOf* , que nos permite montar um *Array* de valores variáveis. Por último a classe é destruída e liberada da memória (**Linha 09**). Salve seu projeto e de o nome a esta *Unit* de *untHeranca.pas*.

Criaremos um exemplo de consulta para demonstrar o funcionamento. Para isso crie um novo formulário acessando *File>New>Other>Inheritable Items>frmHeranca*. Renomeie o novo formulário para *frmCadUsuarios* e salve-o com o nome de *untCadUsuarios.pas*. Selecione a aba de consulta e no *cmbPesquisa* insira na propriedade *Items* o conteúdo de *Código* e *Nome* respectivamente. Altere a propriedade *ItemIndex* para *0* e na aba cadastro adicione os *DBEdit's* referentes aos campos. Ligue-os ao *dsrCadastro* através da propriedade *DataSource* e por último digite seus respectivos *DataField's*. Voltemos para o *frmPrincipal* para implementar as chamadas.

Neste, adicione as *Unit's untLogin*, *untDMPrincipal* e *untCadUsuarios*. Crie uma propriedade chamada *ConfiguraPermissao* conforme mostrado a seguir, e novamente pressione *Ctrl + Shift + C* para que os corpos sejam criados automaticamente:

```
property ConfiguraPermissao : Boolean;
```

Altere o método *SetConfiguraPermissao* conforme o seguinte código:

```
FConfiguraPermissao := Value;
Cadastro1.Enabled   := FconfiguraPermissao;

Consultas1.Enabled  := not FConfiguraPermissao;
```

Vale a pena lembrarmos que *Cadastro1* e *Consultas1* é o nome do *Item* de *Menu* do *MainMenu*. Criaremos agora a chamada da tela de *Login* no evento *OnCreate* do formulário conforme mostra a **Listagem 6.**

O código da **Listagem 6** como podemos observar é bem simples. O formulário é criado dinamicamente (**Linha 03**), em seguida verificamos se o resultado do *ShowModal*, que refere-se ao *ModalResult* do formulário de *Login*, e compara se este é *mrOk* (**Linha 06**). Se esta condição for satisfeita a aplicação será encerrada, caso contrário o método *ConfiguraPermissao* recebe a variável instanciada no servidor de aplicação, configurando assim o acesso de permissões no menu (**Linhas 07** a **09**) . E, finalmente o formulário é destruído e liberado da memória como mostra a **Linha 11.**

A última configuração que mostraremos é a chamada do formulário de cadastro de usuários que deve ser implementada na ação *OnExecute* da *actUsuario* (*ActionList*). O código se encontra detalhado na **Listagem 7.**

Na última listagem do nosso artigo, podemos observar a semelhança com a **Listagem 6**. Novamente na Linha 03 o formulário é criado dinamicamente. Entre as **Linhas 06** e **08** temos a atribuição das propriedades padrões, implementadas no nosso formulário de Herança. Por último o formulário é exibido, e destruído quando necessário (**Linhas 09** e **11**).

Vá ao menu *Project>Options>Forms*, e configure sua aplicação para que crie os formulários na sequência exibida na **Figura 11**

Para finalizar, execute seu sistema e efetue um *login* com

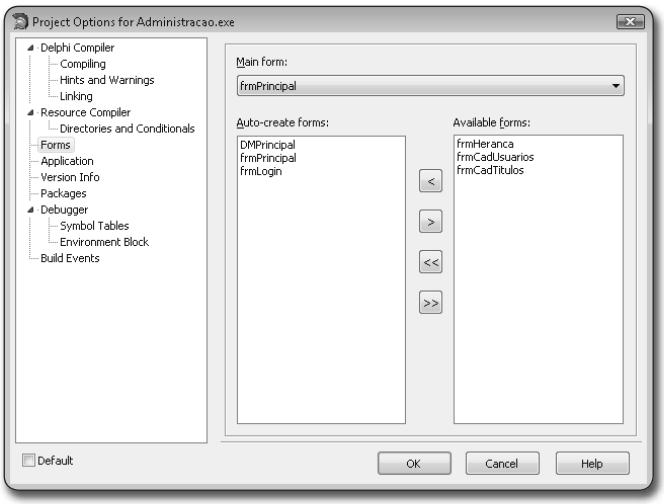


Figura 11. Ordem de criação dos formulários

Listagem 6. Chamada do formulário de Login

```
01  procedure TfrmPrincipal.FormCreate(Sender: TObject);
02  begin
03    frmLogin := TfrmLogin.Create(Self);
04    with frmLogin do
05      try
06        if ShowModal <> mrOk then
07          Application.Terminate
08        else
09          ConfiguraPermissao := DMPrincipal.FAdm;
10      finally
11        FreeAndNil(frmLogin);
12      end;
13  end;
```

Listagem 7. Chamada do formulário de Usuários

```
01  procedure TfrmPrincipal.actUsuarioExecute(Sender: TObject);
02  begin
03    frmCadUsuarios := TfrmCadUsuarios.Create(Self);
04    with frmCadUsuarios do
05      try
06        DataSetCadastro := DMPrincipal.cdsUsuarios;
07        Administrador   := DMPrincipal.FAdm;
08        TipoTela        := 1;
09        ShowModal;
10      finally
11        FreeAndNil(frmCadUsuarios);
12      end;
13  end;
```

o usuário *asantos* e senha *123456*. Ao iniciar, você perceberá que as opções de consulta não estão acessíveis para este usuário devido ao seu nível de permissão. Saia do sistema e faça novamente o *login*, agora com o usuário *gpauli* e senha *889900*. Desta vez o resultado será o inverso, como mostra a **Figura 12**.

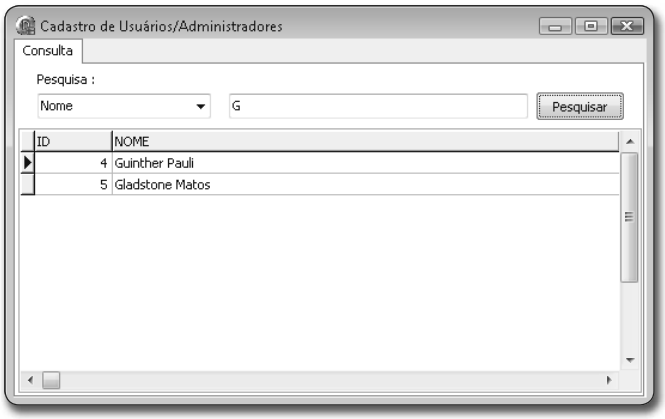


Figura 12. Consulta de Usuários/Administradores com Login limitado

Portabilidade

Portar uma aplicação tradicional para esta nova arquitetura é uma opção interessante, mas não obrigatória. É perfeitamente possível utilizarmos os servidores e clientes tradicionais *DataSnap*, porém, ao migramos, estaremos desfrutando de inúmeras vantagens em relação ao *DataSnap* tradicional.

Conclusão

Neste artigo, foi abordado, de forma sucinta, o *DataSnap 2009*, a nova tecnologia para desenvolvimento multicamadas presente na mais nova versão do *Delphi*.

Como podemos ver, essa novidade simplificou bastante a construção desses tipos de sistemas (*multi-tier*), encorajando até mesmo aqueles desenvolvedores menos experientes. Para aqueles que já possuem alguma experiência nesta arquitetura, pode-se perceber que não foram necessárias grandes modificações em ambas aplicações. Esperamos que todos tenham gostado do artigo, um forte abraço e até a próxima! ●

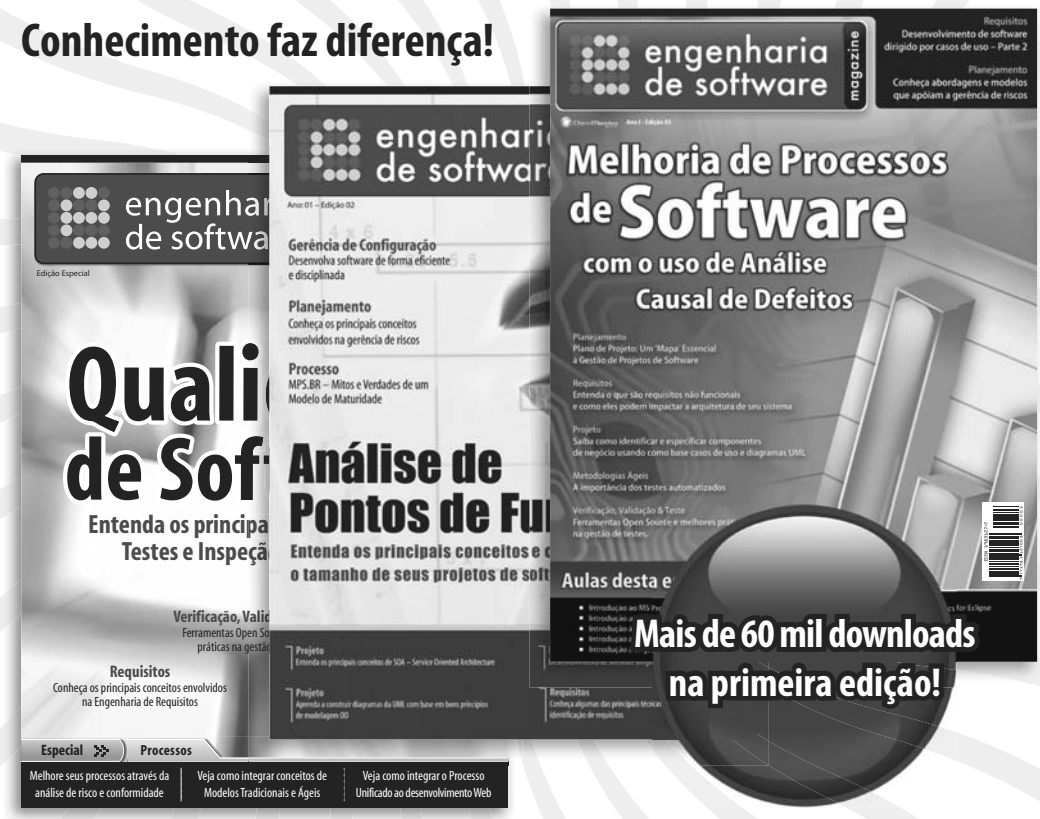
Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Conhecimento faz diferença!



Faça já sua assinatura digital! | *www.devmedia.com.br/es*

Faça um *up grade* em sua carreira.

Em um mercado cada vez mais focado em qualidade ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre outros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lança para os desenvolvedores brasileiros sua primeira revista digital totalmente especializada em Engenharia de Software. Todos os meses você irá encontrar artigos sobre Metodologias Ageis; Metodologias tradicionais (*document driven*); ALM (*application lifecycle management*); SOA (aplicações orientadas a servicos); Analise de sistemas; modelagem; Métricas; orientação à objetos; UML; testes e muito mais. **Assine já!**

DevMedia GROUP

Nesta seção você encontra artigos para iniciantes na linguagem Delphi

Criando aplicações com histórico em XML

Saiba como reaproveitar as últimas consultas do seu banco de dados

Resumo do DevMan

Cada vez mais as empresas e equipes de desenvolvimento almejam por qualidade de software. E um dos quesitos que sempre é posto em pauta, é relacionado diretamente ao desempenho do sistema. E como se sabe, muitos desenvolvedores ainda pecam nesse quesito, principalmente se tratando de consultas. Neste artigo será mostrada uma solução elegante que diminui o tráfego de consultas na rede utilizando a tecnologia DataSnap e que consequentemente é uma alternativa para diminuição de futuros problemas em consultas, podendo aumentar a performance do seu software.

Nesse artigo veremos

- Modelagem e criação do banco de dados de exemplo;
- Conceitos básicos de boas práticas;
- Criação de uma aplicação que faça reutilização de consultas.

Qual a finalidade

Reaproveitar as últimas consultas ao banco de dados, diminuindo o tráfego desnecessário.

Quais situações utilizam esse recurso?

Qualquer software que trabalhe com a tecnologia DataSnap pode utilizar destes recursos.

Daniel Sobrinho Laporte
daniel.laporte@gmail.com
Editor técnico da revista ClubeDelphi. Graduado em sistemas de informação e pós-graduado em administração de bancos de dados pela Universidade de Franca. É analista de sistemas e desenvolvedor Delphi desde 2005 desenvolvendo aplicações cliente/servidor e multicamadas em Delphi com Firebird.

No desenvolvimento de uma aplicação, seja de pequeno, médio ou grande porte, a implementação de consultas é obrigatória, porém, quantas vezes por dia um usuário comum realiza uma mesma consulta simples no seu software, buscando por dados de documentos ou localização por exemplo?

Como sabemos, para cada consulta, é aberta uma transação no banco de dados, que gera de alguma forma um tráfego muitas vezes repetitivo e desnecessário. Uma solução que é comumente adotada por muitos desenvolvedores, é disparar uma consulta na abertura da tela de consulta de modo que sejam trazidos todos os dados, ou grande parte deles de uma única vez, fazendo com que seu carregamento seja lento e causando justamente este tipo de tráfego desnecessário, visto que muitas vezes a informação a se saber em muitos casos é mínima.

No presente artigo, iremos desenvolver uma aplicação de agenda utilizando banco de dados, e que aborda de forma prática uma alternativa para diminuir o tráfego desnecessário, utilizando recursos de carregamento em memória e *XML* presentes na tecnologia *DataSnap.*

Levantando os requisitos necessários

Para o desenvolvimento de nossa aplicação iremos utilizar o *RAD Studio 2009*, juntamente com o banco de dados *Firebird* em sua versão 1.5, porém esta o Delphi trabalha com a versão cinco do *framework dbExpress* que infelizmente não possui um driver nativo para a conexão com o *Firebird*. Sendo assim, utilizaremos uma versão de driver free, criada pelo desenvolvedor *Thiago Borges de Oliveira* para o desenvolvimento da nossa aplicação. O download do driver está disponível no link: *http://www.tbosystems.bluehosting.com.br/dbx4/tbodbxfb_install_0_9_d2009.exe*.

Instalando o driver de conexão

Após ter realizado o *download*, iremos proceder com a instalação do *driver* através do auto-instalador. Execute-o, e na primeira tela clique no botão *next*, na segunda tela selecione o diretório de instalação de sua preferência e em seguida avance novamente. Por último, clique em *Install* para que o processo seja concluído. Se tudo ocorrer bem o seu resultado será o mesmo da **Figura 1.**

Figura 1. Finalização da instalação do driver

Agora que finalizamos a instalação do *driver* a ser utilizado na aplicação, passaremos para o processo de modelagem e criação do banco de dados.

Nota

Vou detalhar alguns processos necessários para a conexão entre o banco de dados e o DBExpress mais adiante, porém, é altamente recomendável que visitem a página do autor disponível no link: http://www.tbosystems.bluehosting.com.br/dbx4/ para assim tomarem nota dos demais requisitos, relativos a distribuição por exemplo.

Nota do DevMan

XML é a abreviação da expressão eXtensible Markup Language. Esta é denominada como linguagem de marcação e trabalha baseada em TAGS. Mais informações sobre esta linguagem e padrões detalhados podem ser encontrados no site: *http://www.w3.org/XML*

Modelando e criando o banco de dados

Como estamos desenvolvendo uma aplicação relativamente pequena, iremos criar três tabelas no nosso banco de dados. Observe o nosso exemplo proposto na **Figura 2**.

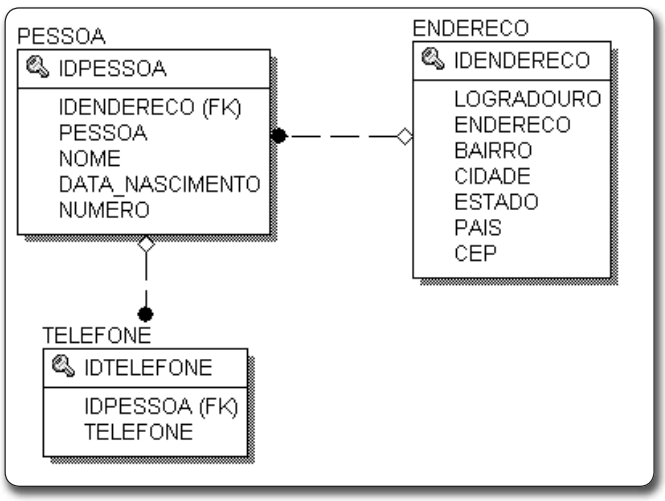


Figura 2. Modelagem do sistema

Começaremos descrevendo a tabela de *ENDERECO* que possui dados comuns a qualquer localização, no entanto, é possível percebermos que o número do endereço (pertencente à residência) não se encontra nesta tabela e sim na tabela *PESSOA*. A razão para tal fato, é que deste modo podemos incluir o mesmo endereço para qualquer contato cadastrado, bastando apenas mudar o número da residência e evitando assim, a necessidade de termos que recadastrá-lo caso haja uma residência na mesma localização e é claro, com numeração diferenciada.

A tabela *PESSOA* em si, refere-se aos contatos que serão cadastrados, portanto trata-se da própria agenda, e será o principal objeto do banco de dados, referenciando a tela principal do nosso exemplo. E por último a tabela *TELEFONE* será filha da tabela *PESSOA*, possibilitando assim a inclusão de diversos telefones para um mesmo contato.

Para a criação do nosso banco de dados, utilize um administrador de sua preferência como o *FlameRobin*, *IBExpert* ou *IBOConsole* para facilitar sua criação através do *script* de *metadata.* O script que utilizaremos para a criação do banco de dados encontra-se na **Listagem 1** conforme pode ser observado.

Configurando necessidades extras

Antes de prosseguir com o desenvolvimento, devemos nos atentar ao fato de algumas configurações extras que devem

ser consideradas. Conforme havia citado na anteriormente, é fundamental ler as notas do autor referentes ao *driver*. No entanto, para quem não observou, segue uma dica dada pelo mesmo. Como estamos utilizando a versão *1.5* do *Firebird*, necessitamos executar um *script* da *UDF TRIM* (conforme mostrado na listagem a seguir) no banco de dados, pois esta funcionalidade só é nativa a partir da versão *2.0*.

```
DECLARE EXTERNAL FUNCTION TRIM
CSTRING(255)
RETURNS CSTRING(255) FREE_IT
ENTRY_POINT 'IB_UDF_ltrim' MODULE_NAME 'ib_udf';
```

Finalizada esta alteração no banco de dados, seguiremos para a modificação necessária na nossa IDE. Então siga abrindo seu *RAD Studio 2009*. É importante atentar que esta configuração não é obrigatória, mas caso você realize a conexão sem seguir estes passos, o seu ambiente ficará exibindo repetidamente a tela de *CPU* a seguir, conforme mostra a **Figura 3.**

Listagem 1. Script do banco de dados

```
SET SQL DIALECT 3;
SET NAMES WIN1252;

SET CLIENTLIB '<DIRETÓRIO DA INSTALAÇÃO DO FIREBIRD>\BIN\
fbclient.dll';

CREATE DATABASE '<DIRETÓRIO DO BANCO DE DADOS>\AGENDA.FDB'
USER 'SYSDBA' PASSWORD 'masterkey'
PAGE_SIZE 4096

DEFAULT CHARACTER SET WIN1252;

CREATE TABLE PESSOA
(IDPESSOA INTEGER    NOT NULL,
 PESSOA VARCHAR(100) NOT NULL,
 IDENDERECO          INTEGER,
 DATA_NASCIMENTO     DATE,
 NUMERO              INTEGER);

ALTER TABLE PESSOA ADD CONSTRAINT PK_PESSOA
PRIMARY KEY(IDPESSOA)
USING INDEX IDX_PESSOA;

CREATE TABLE ENDERECO
(IDENDERECO INTEGER     NOT NULL,
 LOGRADOURO VARCHAR(20) NOT NULL,
 ENDERECO   VARCHAR(50) NOT NULL,
 BAIRRO     VARCHAR(50) NOT NULL,
 CIDADE     VARCHAR(50) NOT NULL,
 ESTADO     VARCHAR(02) NOT NULL,
 PAIS       VARCHAR(50) NOT NULL,
 CEP        VARCHAR(08) NOT NULL);

ALTER TABLE ENDERECO ADD CONSTRAINT PK_ENDERECO
PRIMARY KEY(IDENDERECO)
USING INDEX IDX_ENDERECO;

ALTER TABLE PESSOA ADD CONSTRAINT FK_PESSOA_ENDERECO
FOREIGN KEY(IDENDERECO) REFERENCES ENDERECO (IDENDERECO)
ON DELETE CASCADE
ON UPDATE CASCADE
USING INDEX IDX_PESSOA_ENDERECO;

CREATE TABLE TELEFONE
(IDTELEFONE INTEGER NOT NULL,
 IDPESSOA   INTEGER NOT NULL,
 TELEFONE   VARCHAR(20) NOT NULL);

ALTER TABLE TELEFONE ADD CONSTRAINT PK_TELEFONE
PRIMARY KEY(IDTELEFONE)
USING INDEX IDX_TELEFONE;

ALTER TABLE TELEFONE ADD CONSTRAINT PK_TELEFONE_PESSOA
FOREIGN KEY(IDPESSOA) REFERENCES PESSOA
ON DELETE CASCADE
ON UPDATE CASCADE
USING INDEX IDX_TELEFONE_PESSOA;

CREATE GENERATOR GEN_IDENDERECO;
CREATE GENERATOR GEN_IDPESSOA;
CREATE GENERATOR GEN_IDTELEFONE;
```

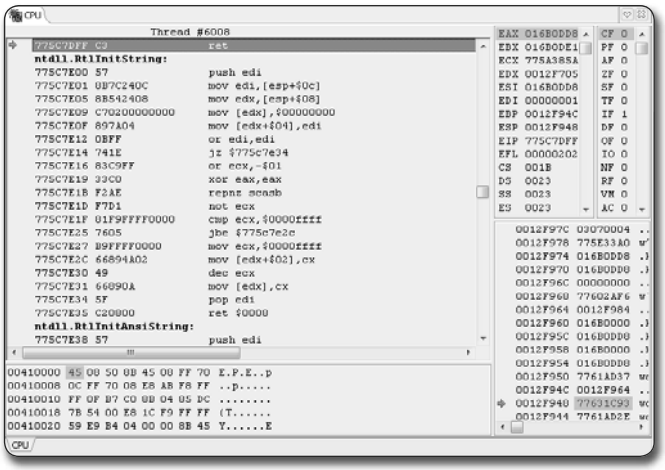


Figura 3. Tela de CPU após a conexão

Para contornar este problema, acesse o menu *Tools>Options> Debugger Options>CodeGear Debuggers* e do lado direito marque a caixa *Ignore non-user breakpoints* conforme mostra a **Figura 4.**

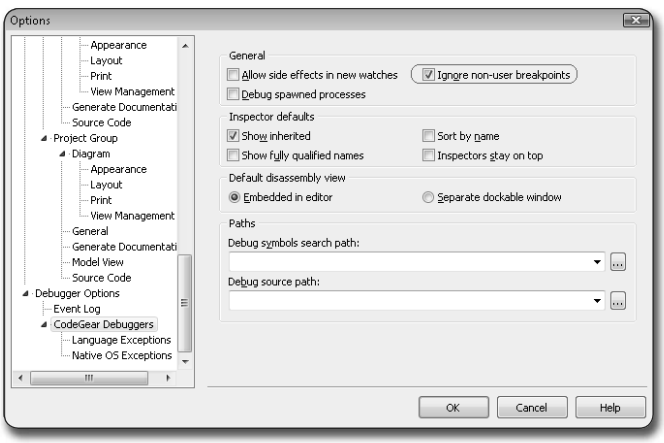


Figura 4. Configuração de opção no RAD Studio 2009

Feito isso teremos finalizado as alterações necessárias no banco de dados e IDE, e agora podemos prosseguir para o desenvolvimento, criando a nossa aplicação de exemplo.

Desenvolvendo a aplicação de exemplo

Agora que estamos com as nossas necessidades devidamente configuradas, partiremos para a parte da criação da aplicação. Em seu *RAD Studio 2009*, direcione-se ao menu *File>New>VCL Forms Application* para criar uma nova aplicação *Win32* e por enquanto no novo formulário aberto, apenas modifique as propriedade *Name* para *frmHeranca* e seu *Caption* para *Herança*. Salve seu projeto clicando no botão *Save All (Shift + Ctrl + S)* ou no menu *File>Save All* definindo o nome da *Unit* como *untHeranca.pas* e o projeto como *Agenda.dproj*. Em seguida faremos a criação do *Data Module* da aplicação, portanto vá em *File>New>Other>Delphi Files* e do lado direito selecione *Data Module*. Altere sua propriedade *Name (DMPrincipal)* e em seguida adicione um *SQLConnection* (*sqlConexao*), cinco *SQLDataSet's* (*sqlPessoa, sqlTelefone, sqlEndereco, sqlPesquisaPessoa, sqlPesquisaEndereco*), quatro

DataSetProvider's (*dspMestreDetalhe, dspEndereco, dspPesquisaPessoa, dspPesquisaEndereco*), cinco *ClientDataSet's (cdsPessoa, cdsTelefone, cdsEndereco, cdsPesquisaPessoa, cdsPesquisaEndereco)* e por último um *DataSource (dsrMestreDetalhe).* Passaremos agora para a configuração da conexão.

Na versão do *Delphi 2009*, diferentemente das anteriores, a conexão no *SQLConnection* não é configurada através de um duplo clique no próprio componente , mas sim através do *Data Explorer* que encontra-se junto ao *Project Manager* (menu *View>Project Manager* ou *Ctrl + Alt + F11*) e que após a instalação se encontrará do mesmo modo da **Figura 5.**

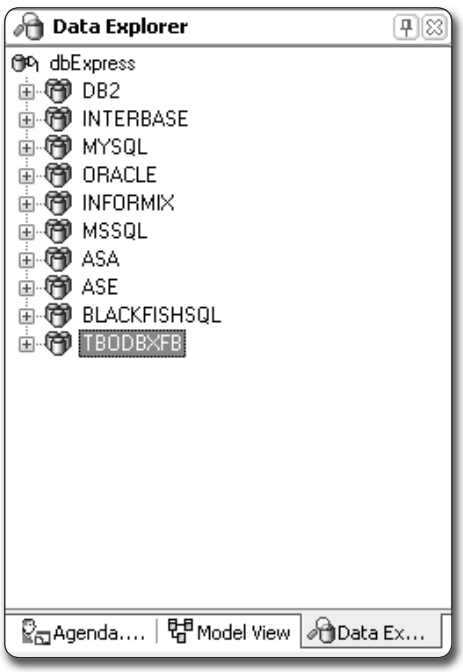


Figura 5. Janela do Data Explorer

Clique com o botão direito sobre o *TDBODBXFB* e selecione a opção *Add New Connection*, na janela a seguir informe o *Connection Name Agenda* e clique em *OK*. Expandindo sua ramificação, perceba que foi adicionada uma conexão, porém sem configuração alguma. Clique sobre a conexão e com o botão direito selecione a opção *Modify Connection* e em seguida clique no botão *Advanced*. Será exibida uma janela para a configuração do banco de dados, configure-a conforme mostra a **Figura 6**. Lembrando que o *Password* do banco de dados é *masterkey*.

Após ter finalizado esta etapa, selecione o componente *sqlConexao* e modifique as propriedades *Driver* para *TBODBXFB*, *ConnectionName* para *Agenda*, *LoginPrompt* para *False* e *VendorLib* para *fbclient.dll*. Seguindo estes dados modifique a propriedade *Connected* para *True*, e perceberá que o componente já consegue banco de dados com êxito. Agora que concluímos a conexão com o banco vamos configurar os componentes do *DataModule*

Adicione as instruções *SQL* da **Listagem 2** nos componentes *SQLDataSet* de acordo com seus respectivos nomes.

Após ter adicionado as sentenças *SQL*, selecione todos os *SQLDataSet's* e configure suas propriedades *SQLConnection* apontando-as para o componente *sqlConexao*. Seguiremos configurando agora o relacionamento *mestre-detalhe*. Primeiramente modifique a propriedade *Options* do *dpsMestreDetalhe*, marcando para *True* as opções *[poIncFieldProps, poCascadeDeletes,*

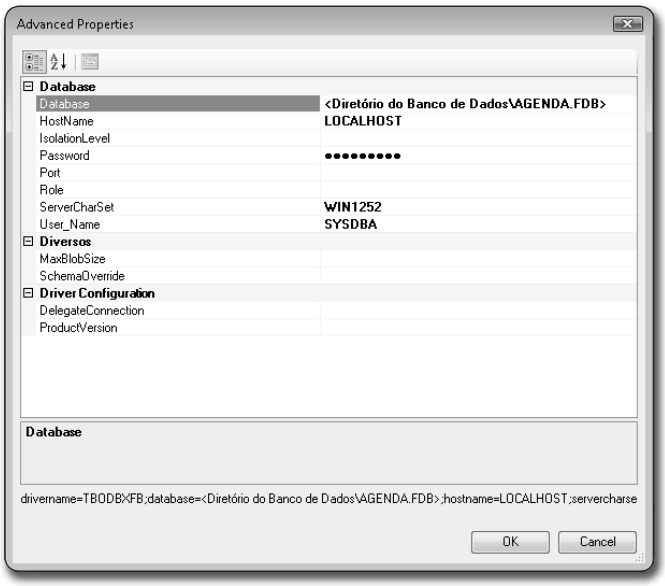


Figura 6. Propriedades avançadas do Data Explorer

Listagem 2. Sentenças SQL dos componentes SQLDataSet

```
[sqlPessoa]
SELECT
  PESSOA.IDPESSOA,
  PESSOA.PESSOA,
  PESSOA.IDENDERECO,
  PESSOA.DATA_NASCIMENTO,
  PESSOA.NUMERO,
  ENDERECO.LOGRADOURO,
  ENDERECO.ENDERECO,
  ENDERECO.BAIRRO,
  ENDERECO.CIDADE,
  ENDERECO.ESTADO,
  ENDERECO.PAIS,
  ENDERECO.CEP
FROM
  PESSOA
LEFT JOIN ENDERECO ON
  PESSOA.IDENDERECO = ENDERECO.IDENDERECO
WHERE
  PESSOA.IDPESSOA =:IDPESSOA

[sqlTelefone]
SELECT
  TELEFONE.IDTELEFONE,
  TELEFONE.IDPESSOA,
  TELEFONE.TELEFONE
FROM
  TELEFONE
WHERE
  TELEFONE.IDPESSOA = :IDPESSOA

[sqlEndereco]
SELECT
  ENDERECO.IDENDERECO,
  ENDERECO.LOGRADOURO,
  ENDERECO.ENDERECO,
  ENDERECO.BAIRRO,
  ENDERECO.CIDADE,
  ENDERECO.ESTADO,
  ENDERECO.PAIS,
  ENDERECO.CEP
FROM
  ENDERECO
WHERE
  ENDERECO.IDENDERECO = :IDENDERECO
```

poCascadeUpdates, poPropagateChanges], a propriedade *UpdateMode* deve ser configurada para *upWhereKeyOnly* e aponte à seu *DataSet* o *sqlPessoa*, para mesma propriedade *(DataSet)* do *dsrMestreDetalhe* ligue também ao *sqlPessoa* e no *sqlTelefone* atribua à propriedade *DataSource* o *dsrMestreDetalhe*. Clique com o botão direito do mouse sobre o *sqlPessoa* e selecione a opção *FieldsEditor,* clique sobre o editor que se abre e também com o botão direito selecione *Add all fields* para que os campos sejam adicionados, configure para cada campo a propriedade *ProviderFlags*, habilitando as opções *[pfInUpdate, pfInWhere, pfInKey]* para o campo relativo a chave primária, *[pfInUpdate]* para os campos que são da própria tabela *(PESSOA)* e para os demais deixe todas as opções desmarcadas pois não sofrerão atualização alguma. Repita este processo de adição de campos e configuração dos *ProviderFlags* nos componentes *sqlTelefone* e *sqlEndereco.*

Para o *dspEndereco* modifique o *UpdateMode* para *upWhereKeyOnly* e em *Options* marque para *True* apenas as propriedades *[poIncFieldProps, poPropagateChanges]*. Nos demais componentes *(dspPesquisaPessoa* e *dspPesquisaEndereco)* apenas habilite em *Options* a propriedade *[poAllowCommandText]* e finalize as configurações dos *DataSetProvider's* atribuindo seus respectivos *DataSet's.*

Selecione agora o *cdsPessoa* e configure sua propriedade *ProviderName* apontando para o *dspMestreDetalhe*, ainda no *ClientDataSet* adicione seus campos do mesmo modo em que fizemos nos *SQLDataSet's* e perceba que temos um campo do tipo *DataSetField* chamado *sqlTelefone*. Este campo será responsável por realizar o relacionamento entre os *ClientDataSet's cdsPessoa* e *cdsTelefone.* Assim sendo, selecione o *cdsTelefone* e modifique a propriedade *DataSetField* para o *cdsPessoasqlTelefone,* deste modo já é possível adicionarmos seus campos no *FieldsEditor.* Finalize adicionando os campos ao *cdsEndereco*.

Nota

Como habilitamos nos DataSetProvider's as opções [poIncFieldProps, poPropagateChanges], ao ativar os ClientDataSet's teremos a propagação automática das configurações relativas a máscaras, DisplayText's e ProviderFlags já definidas nos SQLDataSet's.

Para finalizarmos o nosso *DataModule*, vamos fazer algumas codificações simples. Implemente o código da **Listagem 3** e pressione *Ctrl + Shift + C* para que o cabeçalho seja criado automaticamente.

Listagem 3. Declaração da função GetIDValue

```
01  function TDMPrincipal.GetIDValue(KeyName: String): Integer;
02  var sql : TSQLDataSet;
03  begin
04    try
05      sql := TSQLDataSet.Create(Self);
06      sql.SQLConnection := sqlConexao;
07      sql.CommandText := ' select gen_id(gen_' + KeyName
          + ',1) from
         rdb$database';
08      sql.Close;
09      sql.Open;
10      Result := sql.Fields[0].Value;
11    finally
12      FreeAndNil(sql);
13    end;
14  end;
```

Na **Linha 02** temos a declaração da variável que será o objeto responsável por intermediar a obtenção dos valores dos nossos *Generator's*. Na **Linha 05**, temos a criação do componente em tempo de execução. Perceba que é realizada a conexão com o banco de dados no componente denominado de *sql* (**Linha 06**) e em seguida é atribuída a instrução de busca que dispara automaticamente o *Generator* no banco de dados (**Linha 07**), a conexão é aberta e o resultado obtido é retornado na função (**Linhas 09** e **10)** e por último o objeto *sql* é liberado da memória (**Linha 12**).

Após concluir a implementação da listagem é hora de utilizarmos a nossa função. Logo, selecione o componente *cdsPessoa* e adicione ao evento *AfterInsert* o código da listagem a seguir:

```
cdsPessoaIDPESSOA.Value := GetIDValue('IDPESSOA');
```

Faça o mesmo procedimento também no evento *AfterInsert* do *cdsTelefone* confome exibe a listagem adiante:

```
cdsTelefoneIDTELEFONE.Value := GetIDValue('IDTELEFONE');
```

Por último faça a última implementação no evento *AfterInsert* do *cdsEndereco* conforme é mostrado a seguir:

```
cdsEnderecoIDENDERECO.Value := GetIDValue('IDENDERECO');
```

Assim finalizamos o nosso *DataModule*, salve o seu projeto nomeando a *Unit* como *untDMPrincipal.pas.* Prosseguiremos agora criando as telas de entrada de dados da nossa aplicação.

Quem tem acompanhado meus artigos deve ter percebido que sou adepto do desenvolvimento de herança em minhas aplicações. Como o de costume, iremos fazer o nosso formulário de herança que anteriormente foi criado, mas ainda não sofreu alteração alguma. Abra seu *frmHeranca* e adicione à sua clausula *Uses* a *Unit DBClient*, em seguida, altere suas propriedades *Position* para *poDesktopCenter* e *BorderStyle* para *bsSingle*, faça a modificação das medidas também alterando *Height* para *409* e *Width* para *681*. Comece adicionando um *Panel* (*pnlEsquerdo*) e configurando seu *Align* para *alLeft*, em seguida inclua dentro do *Panel* cinco *BitBtn's (btnIncluir, btnEditar, btnExcluir, btnCancelar, btnSalvar, btnSair)*, caso queira altere a propriedade *Glyph* dos mesmos colocando alguma imagem de sua escolha. Selecione novamente o formulário e adicione um *PageControl (pclPrincipal)* e clicando sobre ele com o botão direito do mouse em *New Page* adicione duas abas definindo a primeira como *tbsPesquisa* e a outra como *tbsCadastro*. Na aba *tbsPesquisa* adicione um *ComboBox (cmbItens)* com a propriedade *Style* configurada como *csOwnerDrawFixed*, um *Edit* (*edtPesquisa*) e um *BitBtn (btnPesquisar)*, adicione dois *DataSource's (dsrPesquisa*, *dsrCadastro)* e por último uma *DBGrid*, modificando suas propriedades *ReadOnly* para *True* e em *Options* habilite *[dgRowSelect],* por fim configure sua propriedade *DataSource* atribuindo o *dsrPesquisa*. O *layout* final de nossa tela pode ser observado na **Figura 7.**

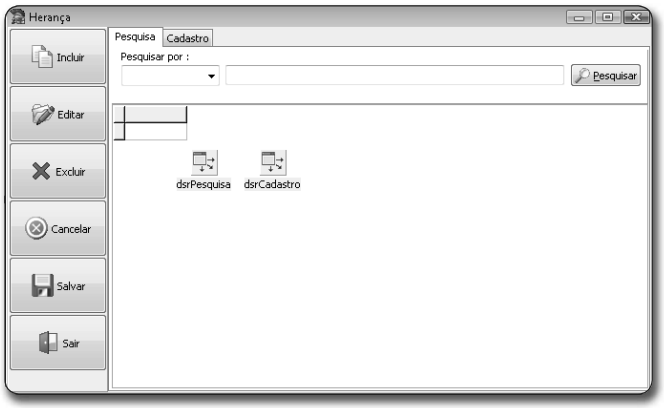


Figura 7. Layout do formulário de herança

Agora que o layout está pronto iremos partir para a codificação. Primeiramente vamos começar definindo duas propriedades que serão responsáveis por atribuir os *ClientDataSet's* aos *DataSource's* então comece definindo o cabeçalho da listagem adiante na sessão *public* do seu formulário e pressione *Ctrl+Shift+C* para a criação dos métodos restantes.

```
property DataSetPesquisa : TClientDataSet;
```

Perceba que automaticamente foi definida uma variável *FDataSetPesquisa* do tipo *TClientDataSet* e uma *procedure SetDataSetPesquisa* que possui um parâmetro do mesmo tipo. Faça uma pequena modificação nesta *procedure* deixando-a como a listagem a seguir:

```
FDataSetPesquisa := Value;
dsrPesquisa.DataSet := FDataSetPesquisa;
```

Em seguida faça o mesmo procedimento, porém, desta vez criando uma propriedade chamada *DataSetCadastro* que também deve ser definida na sessão *public* do seu formulário. O código é semelhante ao da propriedade anterior, como podemos observar:

```
property DataSetCadastro : TClientDataSet;
```

Para finalizar as propriedades, modifique desta vez a *procedure SetDataSetCadastro* deixando-a como a listagem que se segue:

```
FDataSetCadastro := Value;
dsrCadastro.DataSet := FDataSetCadastro;
```

Agora faremos a codificação dos botões iniciando pelo *btnIncluir*, adicione ao evento *OnClick* o código da próxima listagem. Repare que é utilizada a variável *FDataSetCadastro* que foi criada através da propriedade que definimos anteriormente.

```
FDataSetCadastro.Close;
FDataSetCadastro.Open;
FDataSetCadastro.Append;
pclPrincipal.ActivePage := tbsCadastro;
```

Codifique o evento *OnClick* do botão *btnEditar* com o código da **Listagem 4.**

Listagem 4. Evento OnClick do botão de edição

```
01  procedure TfrmHeranca.btnEditarClick(Sender: TObject);
02  begin
03    if FDataSetPesquisa.Active then
04    begin
05      if not FDataSetPesquisa.IsEmpty then
06      begin
07        FDataSetCadastro.Close;
08        FDataSetCadastro.FetchParams;
09        FDataSetCadastro.Params[0].Value :=
            FDataSetPesquisa.Fields[0].Value;
10        FDataSetCadastro.Open;
11        FDataSetCadastro.Edit;
12        pclPrincipal.ActivePage := tbsCadastro;
13      end;
14    end;
15  end;
```

Observando o código da **Listagem 4,** podemos analisar dois testes simples na qual verificam se a variável *FDataSetPesquisa* referenciada pela propriedade *DataSetPesquisa* está ativa e não está vazia (**Linhas 03** a **05)** adiante, nas **Linhas 07** a **11** a variável *FDataSetCadastro* é fechada e os parâmetros são adicionados, e então é passado o valor do campo da pesquisa na posição 0 ao parâmetro 0 referente ao cadastro para que seja realizada a abertura e posteriormente edição do cadastro. Por último é exibida a página de cadastros **(Linha 12).**

Nota

Quando nos referimos as variáveis FDataSetCadastro e FDataSetPesquisa, estamos tratando do conteúdo existente ali dentro que obviamente será um de nossos ClientDataSet's.

Na sequência, é implementado o evento *OnClick* do *btnExcluir* conforme mostrado a seguir na **Listagem 5**:

Listagem 5. Evento OnClick do botão de exclusão

```
01  procedure TfrmHeranca.btnExcluirClick(Sender: TObject);
02  begin
03    if (FDataSetCadastro.Active) and (not FDataSetCadastro.
        IsEmpty) then
04    begin
05      if Application.MessageBox('Deseja excluir ?','Confirmação !',
          MB_YESNO + MB_ICONQUESTION) = ID_YES then
06      begin
07        FDataSetCadastro.Delete;
08        FDataSetCadastro.ApplyUpdates(0);
09        FDataSetPesquisa.Close;
10        edtPesquisa.Clear;
11        pclPrincipal.ActivePage := tbsPesquisa;
12      end;
13    end;
14  end;
```

No evento *OnClick* do botão editar exibido na **Listagem 5** é realizado o mesmo teste que verifica se o *DataSetCadastro* está ativo e possui conteúdo (**Linha 03**), caso seja verdadeira a condição, é solicitada a confirmação ao usuário. Se a operação realmente for confirmada o registro é deletado e gravado no banco de dados (**Linhas 08 e 09**), a consulta é finalizada, o *edtPesquisa* é limpado e a aba de consultas é exibida para que se possa realizar nova consulta.

Seguindo, vamos agora adicionar o evento *OnClick* do *btnCancelar*, que é exibido na listagem a seguir:

```
FDataSetCadastro.Cancel;
FDataSetCadastro.Close;
pclPrincipal.ActivePage := tbsPesquisa;
```

O próximo evento a ser codificado é o *OnClick* do *btnSalvar* que é mostrado na **Listagem 6.**

Listagem 6. Evento OnClick do botão Salvar

```
01  procedure TfrmHeranca.btnSalvarClick(Sender: TObject);
02  begin
03    if Application.MessageBox('Deseja salvar ?',
       'Confirmação !', MB_YESNO
         + MB_ICONQUESTION) = ID_YES then
04    begin
05      FDataSetCadastro.Post;
06      if FDataSetCadastro.ApplyUpdates(0) = 0 then
07      begin
08        Application.MessageBox('Dados salvos!',
           'Confirmação !', MB_OK + MB_ICONINFORMATION);
09        pclPrincipal.ActivePage := tbsPesquisa;
10        FDataSetPesquisa.Close;
11        FDataSetCadastro.Close;
12      end
13      else
14      begin
15        Application.MessageBox('Problemas ao salvar !',
           'Erro !', MB_OK + MB_ICONERROR);
16        FDataSetCadastro.Edit;
17      end;
18    end;
19  end;
```

Realizando uma breve analise na **Listagem 6,** verificamos que é requisitada a confirmação do usuário para a gravação dos dados, sendo verdadeira esta condição, é tentada a gravação (**Linha 06**), e caso seja confirmada é exibida a mensagem de sucesso, a página de pesquisa é exibida e os *DataSet's* são fechados (**Linhas 08** a **11).** Se for retornado algum erro no processo de gravação o método *ApplyUpdates* não retornará o valor 0 e o registro será posto em edição juntamente com a mensagem de erro (**Linhas 15** e **16**).

Por último adicione o evento *OnClick* do *btnSair* como é mostrado a seguir, para assim finalizarmos a codificação dos nossos botões.

```
ModalResult := mrCancel;
Self.Close;
```

Terminada a codificação dos botões, temos ainda alguns controles simples a realizar, vamos começar pelo *cmbItens*. No evento *OnChange* adicione o código a seguir para que a cada vez que for selecionada uma opção o *edtPesquisa* seja limpado.

```
edtPesquisa.Clear;
```

Selecione a *DBGrid* e em seu evento *OnDblClick* adicione o seguinte código que controla o estado da janela:

```
if not FDataSetPesquisa.IsEmpty then
  ModalResult := mrOk
else
  ModalResult := mrCancel;
```

Em seguida, faremos o controle da visibilidade dos botões. Para isso, selecione o componente dsrCadastro e no evento onStateChange atribua o código da listagem a seguir:

```
  btnIncluir.Enabled   := not (FDataSetCadastro.State in [dsInsert,
    dsEdit]);
  btnEditar.Enabled    := (not (FDataSetCadastro.State in
[dsInsert]));
  btnExcluir.Enabled   := (not FDataSetCadastro.IsEmpty) and
   (FDataSetCadastro.Active);
  btnCancelar.Enabled  := FDataSetCadastro.State in [dsEdit,
dsInsert];
  btnSalvar.Enabled    := FDataSetCadastro.State in [dsEdit,
dsInsert];
  btnPesquisar.Enabled := (not (FDataSetCadastro.State in [dsEdit,
    dsInsert]));
```

Estamos quase finalizando nossa herança, mas agora iremos implementar os dois métodos principais, no que dizem respeito ao tema do artigo. Primeiramente crie uma *procedure* na sessão *private* do seu formulário com o código a seguir:

```
  FDataSetPesquisa.SaveToFile(ExtractFilePath(Application.ExeName)
+
    Self.Name + '.xml', dfXML);
```

A *procedure GravaXML*, utiliza o método *SaveToFile* do *ClientDataSet* para gerar um arquivo XML com o conteúdo contido dentro da consulta que realizada. O método *Application.ExeName* retorna o diretório completo de onde se localiza a nossa aplicação, o *ExtractFilePath* faz com que seja indicado apenas o diretório e não o nome do executável, somando ao resultado o nome do formulário e a extensão *.xml*. O segundo parâmetro do método *SaveToFile* especifica qual será o tipo de arquivo que será gerado, no nosso caso um *XML*. Deste modo, se fossemos salvar uma consulta da nossa tela de herança por exemplo, teríamos um arquivo *XML* com o nome de *frmHeranca.xml* localizado no diretório de nossa aplicação.

O segundo método é denominado de *CarregaXML* e sua codificação é exibida na **Listagem 7.**

Listagem 7. Método CarregaXML

```
01  procedure TfrmHeranca.CarregaXML;
02  var
03    Path : String;
04  begin
05    Path := ExtractFilePath(Application.ExeName) +
       Self.Name + '.xml';
06    if FileExists(Path) then
07    try
08      FDataSetPesquisa.Close;
09      FDataSetPesquisa.LoadFromFile(Path);
10    except
11      DeleteFile(Path);
12    end;
13  end;
```

Na **Listagem 7,** observamos que o método *CarregaXML* possui uma variável do tipo *String* denominada de *Path* **(Linha 03).** Assim como o método *GravaXML*, temos a atribuição do nome do arquivo *XML* a esta variável como podemos perceber na **Linha 05**. Assim, é verificado se o arquivo existe no diretório da aplicação. Atendendo a esta condição a pesquisa é fechada e automaticamente carregada para o *ClientDataSet* através do método *LoadFromFile* (**Linhas 08** e **09**). Em caso de exceção o arquivo será excluído **(Linha 11).**

Após termos criado os métodos principais do nosso artigo, simplesmente devemos fazer suas chamadas dentro do formulário de herança, e logo este será propagado a todas as telas

existentes e baseadas na herança. Sendo assim, selecione seu formulário e no evento *OnShow* declare a chamada do método de carregamento do arquivo e controle de visibilidade dos botões conforme é mostrado a seguir:

```
CarregaXML;
dsrCadastroStateChange(Sender);
```

Para finalizarmos a herança, adicione ao evento *OnDestroy* do formulário a chamada do método que realiza a gravação e em seguida fecha a consulta, evitando que os *DataSet's* se mantenham abertos, como mostrado adiante:

```
GravaXML;
FDataSetPesquisa.Close;
```

Com isso finalizamos a herança. Agora é necessário que façamos os demais cadastros da nossa aplicação utilizando como base este formulário. Vá ao menu *File>New>Other>Inheritable Items* e do lado direito selecione o *frmHeranca* para que um novo *Form* seja criado com as implementações baseadas no nosso modelo.

No novo *Form* altere seu *Name* para *frmCadEndereco* e *Caption* para *Endereço*, no componente *cmbItens* modifique a propriedade *Itens* adicionando *Código* e *Endereço*, após isso altere seu *ItemIndex* para *0*. Selecione agora a aba de cadastros e adicione oito *Label's* configurando suas propriedades *Caption* para *Código, Logradouro, Endereço, Bairro, Cidade, Estado, CEP* e *País*, adicione também sete *DBEdit's* e um *DBText*. Selecione todos os *DBEdit's* e o *DBText* e configure suas propriedades *DataSource* apontando para o *dsrCadastro* e manualmente digite os nomes dos respectivos campos na propriedade *DataField* de cada um deles. O resultado você poderá observar na **Figura 8.**

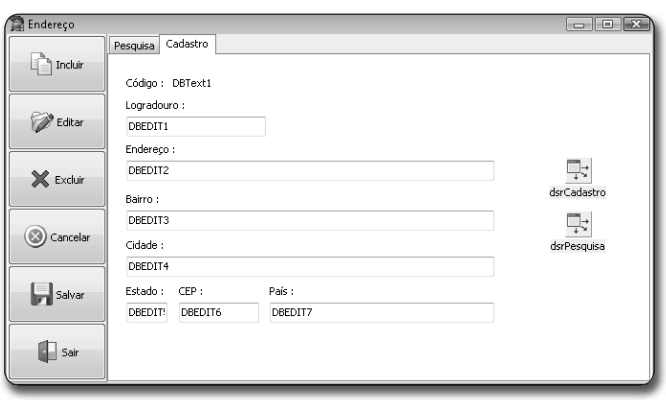


Figura 8. Layout do formulário de cadastro de endereços

A única codificação necessária para este formulário será referente à consulta de dados. Selecione o botão *btnPesquisar* e em seu evento *OnClick* faça a implementação da **Listagem 8**.

O código da **Listagem 8** é bem simples como podemos ver. Entre as **Linhas 02** a **07** temos a declaração de uma constante denominada *SQL_ENDERECO*, na qual recebe a instrução *SQL* necessária para realizarmos a consulta, porém ainda sem as cláusulas de condição. Em seguida temos a criação de uma variável do tipo *String* **(Linha 09).** Na **Linha 12** temos o teste que verifica se há conteúdo dentro do *edtPequisa*, sendo verdadeira a pesquisa será realizada de acordo com o *ItemIndex* do *cmbItens* que caso tenha valor 0 é realizada a concatenação da constante com a condição de busca por código **(Linha 14)**, caso contrário é realizada a busca por endereço **(Linha 15)**. Por último, **(Linhas 17 a 19)** a consulta é fechada, a instrução é passada ao componente de pesquisa e finalmente aberta, retornando os registros referentes à condição consultada. Assim finalizamos nossa tela de cadastro de endereços, passaremos para o código da agenda em si. Salve seu projeto nomeando a nova *Unit como untCadEndereco.pas.*

Novamente acesse o menu *File>New>Other>Inheritable Items* e selecione o *frmHeranca* para que outro formulário seja criado. Primeiramente altere seu *Name* para *frmCadAgenda*, seu *Caption* para *Agenda*, adicione à cláusula *Uses* a *unit DBClient* e através do menu *File>Use Unit* adicione também as *units untDMPrincipal, untCadEndereco.* Selecione o componente *cmbItens* e adicione à sua propriedade *Items* os textos *Código* e *Pessoa*¸ e por último altere sua propriedade *ItemIndex* para o valor *0*. Em seguida vamos adicionar os componentes necessários. Inclua ao seu novo *Form* um *MainMenu* e dê um duplo clique sobre ele para adicionarmos os menus da aplicação. O primeiro será chamado de *Opções* e dentro deste criaremos o sub-item *Consultar Endereços*, ligue este *MainMenu* à propriedade *Menu* do seu *Form*. Agora adicione um *DataSource (dsrTelefone)*, doze *Label's (Código, Data de Nascimento, Pessoa, Logradouro, Endereço, N , Bairro, Cidade, Estado, CEP, País* e *Telefone a inserir*), oito *DBText's* e altere o nome de um deles para *dbedTelefone*, um *DBNavigator (dbnTelefone)* e uma *DBGrid*. Feito isso, selecione todos os *DBEdit's* e os *DBText's* e modifique suas propriedades *DataSource* apontando para o *dsrCadastro*, novamente configure seus *DataField's* aos respectivos campos como feito no cadastro anterior, ligue a *DBGrid* e o *DBNavigator* ao *dsrTelefone* através da propriedade *DataSource* e por fim adicione dois

Listagem 8. Evento OnClick do botão Pesquisar no cadastro de Endereços

```
01  procedure TfrmCadEndereco.btnPesquisarClick(Sender: TObject);
02  const
03    SQL_ENDERECO =
04    ' SELECT ENDERECO.IDENDERECO, ENDERECO.LOGRADOURO, '     + #13 +
05    ' ENDERECO.ENDERECO, ENDERECO.BAIRRO, ENDERECO.CIDADE, ' + #13 +
06    ' ENDERECO.ESTADO, ENDERECO.PAIS, ENDERECO.CEP '         + #13 +
07    'FROM ENDERECO ';
08  var
09    stParametro : String;
10  begin
11    if Trim(edtPesquisa.Text) <> '' then
12    begin
13      case cmbItens.ItemIndex of
14     0 : stParametro := SQL_ENDERECO + ' WHERE ENDERECO.IDENDERECO = '  +
          edtPesquisa.Text;
15     1 : stParametro := SQL_ENDERECO + ' WHERE ENDERECO.ENDERECO LIKE ' +
          QuotedStr(edtPesquisa.Text) + ' || ' + QuotedStr('%');
16      end;
17      DataSetPesquisa.Close;
18      DataSetPesquisa.CommandText := stParametro;
19      DataSetPesquisa.Open;
20    end
21    else
22     Application.MessageBox('Digite algo !', 'Erro !',
        MB_ICONERROR + MB_OK);
23  end;
```

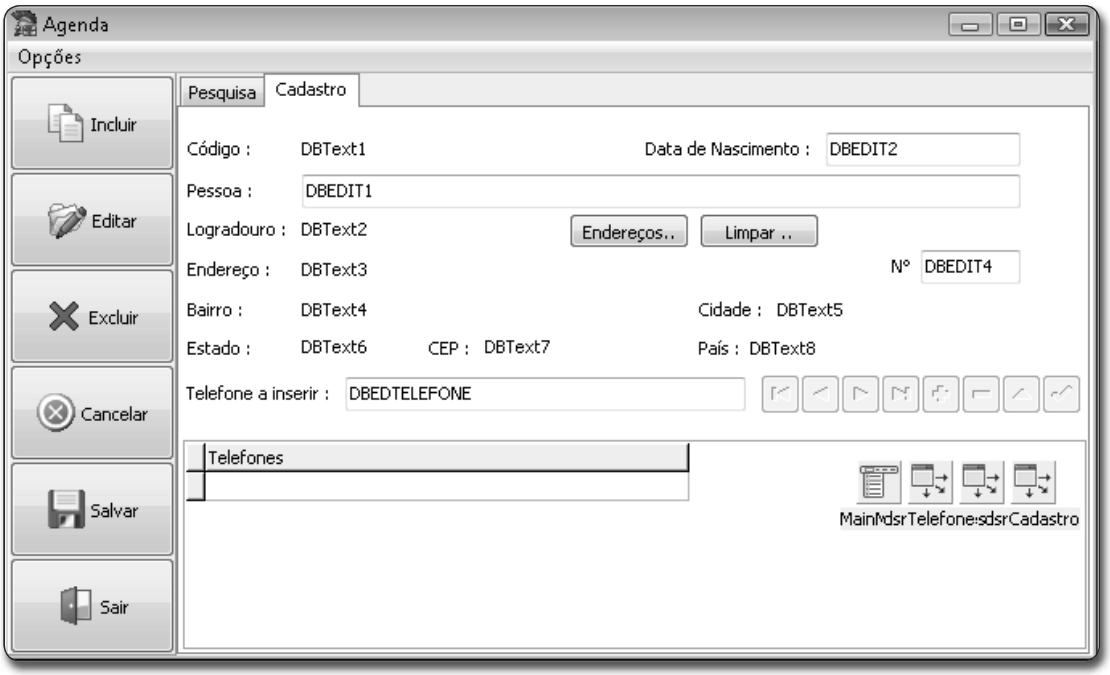


Figura 9. Layout final do formulário de Agenda

SpeedButton's (spbEnderecos, spbLimpar). O resultado final você observa na **Figura 9.**

Iremos agora finalizar nossa aplicação configurando todas as codificações restantes. Declare novamente uma propriedade chamada DataSetTelefone como a implementação a seguir, e pressione as teclas *Ctrl* + *Shift* + *C* para que os métodos restantes sejam definidos automaticamente.

```
property DataSetTelefone : TClientDataSet;
```

Deste modo, como já havíamos comentado será criado o método *SetDataSetTelefone* que é uma *procedure*, altere-a deixando como a seguir:

```
  FDataSetTelefone := Value;
  dsrTelefone.DataSet := FDataSetTelefone;
```

No evento *OnCreate* do seu formulário faça a seguinte implementação para atribuir os *ClientDataSet's* às suas respectivas propriedades:

```
  inherited;
  DataSetPesquisa := DMPrincipal.cdsPesquisaPessoa;
  DataSetCadastro := DMPrincipal.cdsPessoa;
  DataSetTelefone := DMPrincipal.cdsTelefone;
```

Selecione o *dsrCadastro* e adicione ao seu evento *OnStateChange* a seguinte codificação:

```
  inherited;
  dbnTelefone.Enabled := DataSetCadastro.State in [dsEdit,
dsInsert];
  dbEdTelefone.ReadOnly := not (DataSetCadastro.State in [dsEdit,
    dsInsert]);
```

Faremos agora a chamada ao cadastro de endereços por meio do acesso ao menu. Selecione o componente *MainMenu* e dê um duplo clique sobre ele. No sub-item *Consultar Endereços* adicione o evento *OnClick* da **Listagem 9**.

Listagem 9. Evento OnClick do sub-item de menu consultar endereços

```
01  procedure TfrmCadAgenda.ConsultarEndereos1Click(Sender: TObject);
02  begin
03    inherited;
04    frmCadEndereco := TfrmCadEndereco.Create(Self);
05    with frmCadEndereco do
06      try
07        DataSetPesquisa := DMPrincipal.cdsPesquisaEndereco;
08        DataSetCadastro := DMPrincipal.cdsEndereco;
09        ShowModal;
10      finally
11        FreeAndNil(frmCadEndereco);
12      end;
13  end;
```

Listagem 10. Evento OnClick do btnPesquisar

```
01  procedure TfrmCadAgenda.btnPesquisarClick(Sender: TObject);
02  const
03    SQL_PESSOA = ' SELECT PESSOA.IDPESSOA, PESSOA.PESSOA FROM PESSOA ';
04  var
05    stParametro : String;
06  begin
07    if Trim(edtPesquisa.Text) <> '' then
08    begin
09      case cmbItens.ItemIndex of
10      0 : stParametro := SQL_PESSOA + ' WHERE PESSOA.IDPESSOA = '  +
          edtPesquisa.Text;
11      1 : stParametro := SQL_PESSOA + ' WHERE PESSOA.
          PESSOA LIKE ' +
          QuotedStr(edtPesquisa.Text) + ' || ' + QuotedStr('%');
12      end;
13      DataSetPesquisa.Close;
14      DataSetPesquisa.CommandText := stParametro;
15      DataSetPesquisa.Open;
16    end
17    else
18      Application.MessageBox('Digite algo !', 'Erro !',
          MB_ICONERROR + MB_OK);
19  end;
```

Como pudemos ver na **Listagem 9**, apenas foi realizada a criação do formulário de cadastro de endereços em tempo de execução (**Linha 04**), a configuração das propriedades necessárias

(**Linhas 07** e **08**) a exibição do formulário e finalmente a destruição do objeto (**Linhas 09** e **11** respectivamente).

O próximo passo é implementarmos a consulta da nossa agenda. Para isso selecione o botão *btnPesquisar* e adicione ao evento *OnClick* o código da **Listagem 10.**

Selecione o botão *spbLimpar* e em seu evento *OnClick* faça a implementação da **Listagem 11.**

Nota

Não entrarei em detalhes sobre a Listagem 10, pois esta segue a mesma linha de raciocínio da Listagem 8 anteriormente comentada.

Listagem 11. Evento OnClick do SpeedButton spbLimpar

```
01  procedure TfrmCadAgenda.spbLimparClick(Sender: TObject);
02  var i : Integer;
03  begin
04    if (FDataSetCadastro.State in [dsEdit, dsInsert]) then
05    begin
06      for i := 0 to FDataSetCadastro.FieldCount - 1 do
07        if ([] = FDataSetCadastro.Fields[i].ProviderFlags) then
08          FDataSetCadastro.Fields[i].Clear;
09    end;
```

Na **Listagem 11** temos a declaração de uma variável local de tipo inteiro (**Linha 02**) que será responsável por realizar o controle de contagem dos campos. Na **Linha 04** é testado o estado do *DataSet* de cadastro e caso esteja em estado de edição ou inserção a contagem de campos será iniciada **(Linha 06)**. Se você configurou as propriedades *ProviderFlags* dos campos corretamente, todos os campos que não fazem parte da cláusula de atualização do *DataSetProvider* estarão com todas as opções desmarcadas e consequentemente serão automaticamente limpados (**Linhas 06 a 08**).

E para finalizar a codificação, iremos fazer o último controle da nossa aplicação que é relacionado ao evento *OnClick* do *spbEnderecos* e que terá por objetivo retornar os campos de endereço no ato da consulta a este formulário. Acompanhe sua codificação no código da **Listagem 12.**

Na **Listagem 12** temos primeiramente a definição das variáveis que servirão de manipulação e controle em nossa rotina (**Linhas 02** a **04),** realiza-se então o teste que identifica se o cadastro atual está em estado de inserção ou edição (**Linha 06**), sendo satisfeita esta condição, o cadastro de endereços é criado em tempo de execução tendo configuradas suas propriedades necessárias **(Linhas 08** a **11).** Perceba na **Linha 12** que a variável *boControle* recebe sua atribuição de valor de acordo com o retorno do *ModalResult* obtido no cadastro de endereços. Sendo ele *mrOk* ela será setada como *True*, caso contrário será *False*. Em seguida é realizada é realizado um laço na contagem de campos da pesquisa através do comando *for* (**Linha 13**), o mesmo laço é definido contando os campos do *DataSet* de cadastro do nosso *frmAgenda*. Note na **Linha 15** que estamos utilizando a variável *j* para o controle do *DataSetCadastro*. Na **Linha 17** temos a verificação do nome do campo do *DataSet* de pesquisa *(Cadastro de endereços)* através de sua posição em

Listagem 12. Evento OnClick do SpeedButton spbEnderecos

```
01  procedure TfrmCadAgenda.spbEnderecosClick(Sender: TObject);
02  var
03    i, j : Integer;
04    boControle : Boolean;
05  begin
06    if FDataSetCadastro.State in [dsEdit, dsInsert] then
07    begin
08      frmCadEndereco := TfrmCadEndereco.Create(Self);
09      try
10        frmCadEndereco.DataSetPesquisa := DMPrincipal.
           cdsPesquisaEndereco;
11        frmCadEndereco.DataSetCadastro := DMPrincipal.
           cdsEndereco;
12        boControle := frmCadEndereco.ShowModal = mrOk;
13        for i := 0 to frmCadEndereco.DataSetPesquisa.
           FieldCount - 1 do
14        begin
15          for j := 0 to Self.DataSetCadastro.FieldCount - 1 do
16          begin
17        if UpperCase(frmCadEndereco.DataSetPesquisa.Fields[i].
           FieldName) =UpperCase(Self.DataSetCadastro.Fields[j].
            FieldName) then
18            begin
19              if boControle then
20                Self.DataSetCadastro.Fields[j].Value :=
                    frmCadEndereco.DataSetPesquisa.Fields[i].Value
21              else
22                Self.DataSetCadastro.Fields[j].Clear;
23            end;
24          end;
25        end
26      finally
27        FreeAndNil(frmCadEndereco);
28      end;
29    end;
30  end;
```

comparação com os campos referentes ao *DataSet* de cadastro do formulário atual. Se os nomes em suas respectivas posições forem idênticos, a variável *boControle* é testada para que os campos sejam atribuídos (**Linha 20**) ou limpados (**Linha 22**) e por último é realizada a destruição do objeto da memória (**Linha 27**).

Com isso finalizamos toda a codificação necessária. Direcione-se ao menu *Project>Options>Forms* e em *Auto-create forms* mantenha apenas o *DMPrincipal* e o *frmCadAgenda*. Após isto salve seu projeto em *File>Save All* nomeando a *Unit* como *untCadAgenda* para que seu projeto esteja finalizado.

Conclusão

Com isso, pudemos visualizar uma solução eficaz para evitar o tráfego de rede, pois as consultas sempre estarão em disco, sendo persistidas em arquivos *XML* criados automaticamente por nossa aplicação. Conhecer maneiras de obter maior performance das nossas aplicações, sempre são de grande serventia no nosso dia-a-dia, finalizo aqui meu artigo e espero que todos tenham gostado. Um forte abraço e até a próxima. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Nesta seção você encontra artigos sobre a linguagem PHP e a ferramenta Delphi for PHP

Web Services no Delphi for PHP

Compartilhe serviços via web e integre sua aplicação com clientes e fornecedores

Nesse artigo veremos

- O que são Web Services;
- O que é e para que serve o WSDL;
- Como funciona o Protocolo SOAP;
- Como criar e publicar Web Service com Delphi for PHP.

Qual a finalidade

Demonstrar todo o poder e flexibilidade em se oferecer serviços via Web Services, bem como a facilidade em implementá-los no Delphi for PHP.

Quais situações utilizam esse recurso?

Sempre que houver necessidade de se realizar trocas de informações entre diferentes aplicações sem que se conheça como as mesmas foram desenvolvidas e/ou foram implementadas, ou mesmo quando se quer oferecer algum tipo de serviço via Web como por exemplo a consulta de CEP, verificação de situação fiscal, etc.

Rodrigo Carreiro Mourão

rodrigocarreiro@rmfactory.com.br

Apaixonado por tecnologia da informação. Atualmente é sócio diretor da RM Factory Soluções em TI, empresa especializada em consultoria na área de TI e desenvolvimento de softwares customizados, parceira da TDS Tecnologia - RJ. Possui os títulos de BDS2006 Win32 Product Certified e Borland Instructor Certified. Como instrutor de treinamentos oficiais já ministrou treinamentos para centenas profissionais e servidores de órgãos como TJ-RJ, JF, BB, IBGE, MB, EB, etc. Palestrante da Borland Conference 2007. Mantenedor da comunidade www.delphi-sophp.com, a primeira do Brasil 100% feita com Delphi for PHP. Autor de diversos artigos técnicos veiculados nos principais sites e revistas sobre Delphi do Brasil .

Resumo do DevMan

Cada vez mais os Web Services se tornam presentes na vida dos desenvolvedores e por que não das pessoas em geral. Se você ainda não teve a necessidade de desenvolver ou consumir um Web Service saiba que isto pode estar mais perto do que você imagina. Um dos maiores projetos em desenvolvimento e implementação no Brasil é sem dúvida nenhuma o projeto da NF-e (Nota Fiscal Eletrônica). Pela definição oficial, uma nota fiscal eletrônica (NF-e) é um documento de existência apenas digital, emitido e armazenado eletronicamente, com o intuito de documentar uma operação de circulação de mercadorias ou uma prestação de serviços, ocorrida entre as partes. A especificação do projeto, que pode ser vista no site oficial http://www.nfe.fazenda.gov.br/portal, define o uso de padrões abertos de tecnologia, não impondo o uso de nenhuma solução proprietária. Dentre as principais tecnologias envolvidas figuram: Web Services e XML.

Que atire a primeira pedra quem nunca precisou gerar ou ler um arquivo texto em um projeto de integração. Sim, isso é mais comum em desenvolvimento de software que podemos imaginar. Certa vez estava eu em uma empresa, feliz e sorridente tocando minha vida quando meu superior veio dar a notícia de que toda parte de pagamentos e cobranças da empresa seriam automatizadas através de integração com os bancos que trabalhávamos. Para variar os pagamentos dos prestadores de serviços e parceiros eram feitos através de um único banco, já a cobrança. Essa era feita por três bancos diferentes. Isso mesmo: três instituições bancárias.

Pois bem, lá fui eu entrar em contato com os bancos para poder obter a documentação dos sistemas. Em alguns dias minha caixa de email estava repleta de arquivos *.pdf* contendo dezenas de páginas cada um. Era um tal de *CNAB 080* para cá, *CNAB 240* para lá. Regras do *Banco do Brasil* aqui, definições da *Caixa Econômica Federal* ali, enfim! Em pouco tempo tinha tanta documentação para ler que já não fazia mais nada.

Mas o que isso tudo tem haver com *Web Services*? Nada! E este foi todo o problema. Por trabalhar com integração com arquivos texto a *FEBRABAN*, ou melhor, os bancos têm que disponibilizar suas regras e padrões através de extensas documentações. Isso tudo para que seus clientes possam gerar os arquivos dentro do que se espera, e para que a integração possa ser feita de maneira segura e eficiente. Isto toma muito tempo e ainda temos outros inconvenientes. Um arquivo texto não se auto descreve como um arquivo *XML*. Por isso as vastas documentações, para nos informar quem é quem nas milhares de linhas com dezenas de caracteres que temos que enviar e receber nesta integração.

No cenário exposto acima, os *Web Services* cairiam como uma luva. Eles nos poupariam horas de leituras de documentação uma vez que o próprio descritor de serviço (*WSDL*), já nos exibiria os serviços disponíveis, assim como os objetos a serem enviados. Óbvio que a documentação não seria extinta, mas se resumiria apenas às regras do negócio e não em páginas e páginas de *layout* de registros com posição inicial, posição final, tamanho e tipo. Assim, nós estaríamos consumindo um ou mais serviços através dos *Web Services* que as instituições financeiras disponibilizariam para seus clientes. Neste cenário, nós desenvolvedores, ou melhor, as aplicações, estariam atuando como clientes destes *Web Services*. Mas vamos a outro cenário.

Nota do DevMan

O padrão estabelecido pelo CNAB (Centro Nacional de Automação Bancária) foi criado para permitir a troca de arquivos entre bancos e clientes. O objetivo destes arquivos é intercambiar informações digitalmente entre o sistema de informática do banco e o do cliente. Dentre as informações podemos citar: cobrança (boletos bancários), pagamentos, extrato (para conciliação), débito em conta, e custódia de cheques. Cada um destes produtos tem seu fluxo de informação e, portanto um layout. O arquivo contém texto puro e colunas fixas definidas pela FEBRABAN, cada banco, por possuir suas próprias peculiaridades, tem suas variações que são previstas dentro do padrão. Estes arquivos são enviados ou recebidos de diversas formas, mas principalmente através do site do banco.

Ententendo a lógica

Imagine que você trabalha em uma grande empresa, como por exemplo uma editora. A direção da empresa decidiu automatizar o processo de vendas para permitir que seus grandes clientes coorporativos possam efetuar seus pedidos de forma automática, sem a intervenção humana. Não seria ótimo? Reduziria-se assim o tempo de digitação das notas, possíveis falhas humanas no processo, um custo de compra menor para o cliente, uma reposição *Just-in-time* e muito mais. Porém você está em um dilema: como oferecer este tipo de serviço ao cliente? Conceder uma senha para acessar o banco de dados e executar uma instrução de inserção no mesmo? Ou então fazer como os bancos. Vamos criar longas documentações com padrões de arquivos *.txt* e iniciar a brincadeira.

Aqui mais uma vez, os *Web Services* podem nos auxiliar. Que tal disponibilizar para seus clientes um serviço *Web* onde ele possa nos enviar um lote com os produtos que necessita? Toda a informação necessária para o processo será descrita no *WSDL*, cabendo ao cliente simplesmente consumir o serviço e informar os dados, e mais: nosso *Web Service* poderia ainda ter outros recursos que retornassem o *status* de um pedido feito pelo cliente, dados de faturamento e muito mais. Agora, neste cenário a aplicação seria a provedora dos serviços e não mais cliente.

A partir de agora veremos como funciona um *Web Service*. Tomaremos ciência de toda a tecnologia envolvida no processo, conheceremos a arquitetura, protocolo de comunicação e muito mais. Na seqüência, vamos criar nosso primeiro *Web Services* utilizando *Delphi for PHP*, e com isso disponibilizaremos serviços para que nossos clientes possam consumir independente da linguagem que utilizem ou plataforma que desenvolvam. Então vamos nessa!

Entendendo os Web Services

No início da internet, as tecnologias permitiam você conectar-se a um site e baixar o conteúdo deste. O *HTML* permitia exibir a informação para os usuários na grande rede. Porém, novas tecnologias e *frameworks* de desenvolvimento foram surgindo, permitindo assim uma integração maior entre os sistemas na *Web*. Hoje em dia, com todo este crescimento, os sistemas estão realizando tarefas cada vez mais complexas. Surgiu então a necessidade de disponibilizar os serviços destes sistemas através de uma interface de acesso simples e bem definida. Assim, nasceram os *Web Services*.

Qualquer empresa hoje em dia precisa prover serviços, seja para seus clientes, para outra empresa, para os prestadores de serviço, ou outra necessidade qualquer. Seria excelente se esses serviços pudessem ser automatizados. Isso reduziria tempo, dinheiro e eliminaria possíveis erros humanos. Por exemplo, não é necessário existir um vendedor em cada estado do país se seu cliente já possui o preço e outros detalhes para realizar um pedido de compra. Este pedido pode, e já é feito hoje em dia, via *interfaces* computacionais. O cliente entra em seu site, monta o pedido e confirma a compra. Isto é um tipo de serviço *Web*, ou seja, um serviço que está publicado na *Web* para que qualquer pessoa possa utilizar. Este tipo de serviço

atende a pessoa física. Agora leve isto para o relacionamento entre grandes instituições. É muito trabalhoso e caro para uma empresa manter uma pessoa, ou equipe, para entrar em um site, se *logar*, selecionar um a um os produtos e depois fechar o pedido. O ideal é que o sistema desta empresa se integre ao site do fornecedor e possa de uma forma simples e transparente enviar, consultar e cancelar pedidos de compras.

Web Services foram criados para construir aplicações deste tipo. Aplicações que são serviços na internet. Porém não faz parte do conceito de *Web Service* criação de interfaces gráficas para os usuários, deixando esta parte para outras empresas ou pessoas desenvolverem. *Web Service* é a tecnologia para comunicação entre sistemas, sendo muito usada em aplicações *B2B* (*Business to Business*). A comunicação entre os serviços é padronizada, possibilitando a independência de plataforma e de linguagem de programação. Por exemplo, um sistema de comércio eletrônico em *PHP* sendo executado em um servidor *Linux*, pode acessar com transparência o sistema de um ou mais fornecedores desenvolvido em *.Net* e sendo executado em um servidor *Windows* para informar o prazo de um produto mediante o tempo de fabricação informado pelo fabricante.

O *Web Service* faz com que os recursos de uma aplicação estejam disponíveis na internet de uma forma padronizada. Os *browsers* já fazem isso, porém dependem da intervenção humana para consumir os serviços, não sendo eficazes para comunicação entre as aplicações. A grande sacada dos *Web Services* é que eles permitem que aplicações diferentes se comuniquem entre si e consumam recursos umas das outras independente da plataforma. Com eles, uma aplicação pode chamar outra para executar tarefas simples ou complexas, mesmo que as duas aplicações estejam em diferentes sistemas e escritas em diferentes linguagens, ou seja, os *Web Services* fazem com que os seus recursos estejam disponíveis para que qualquer aplicação cliente possa consumir. Os *Web Services* são identificados por um *URI* (*Unique Resource Identifier*), descritos e definidos usando *XML* (*eXtensible Markup Language*) e utilizam o protocolo *SOAP* sobre *HTTP* como transporte entre o cliente e servidor. A **Figura 1** demonstra toda a arquitetura envolvida em um processo que utilize *Web Services*.

Elementos de um Web Service

O *W3C* e *OASIS* são as instituições responsáveis pela padronização dos *Web Services*. Empresas como *IBM* e *Microsoft*, apóiam o desenvolvimento deste padrão. Segundo o *W3C* (*World Wide Web Consortium*) um *Web Service* define-se como: um sistema de software projetado para suportar a interoperabilidade entre máquinas sobre rede. Tem uma relação descritiva em um formato *machine-processable*, especificamente *WSDL* (*Web Service Description Language*). Outros sistemas interagem com o *Web Service* usando as mensagens *SOAP*, tipicamente sobre *HTTP* com *XML*. As bases para a construção de um *Web Service* são os padrões *XML* e *SOAP*. O transporte dos dados é realizado normalmente via protocolo *HTTP* ou *HTTPS* para conexões seguras, os dados são transferidos no formato *XML* e encapsulados pelo protocolo *SOAP*.

Nota

Em TI (Tecnologia da Informação), um Identificador de Recursos Uniforme (URI) (Uniform Resource Identifier (em inglês)) é uma cadeia de caracteres compacta usada para identificar ou denominar um recurso na internet. O principal propósito desta identificação é permitir a interação com representações do recurso através de uma rede, tipicamente a rede mundial, usando protocolos específicos. URI's são identificadas em grupos definindo uma sintaxe específica e protocolos associados.

Definição do WSDL

Se algum dia, alguém chegar para você e lhe disser que terá que construir um cliente para um *Web Service*. Então, a primeira coisa que você irá precisar é o *WSDL*. Principalmente se for construir o cliente com *Delphi Win32* ou *Delphi .Net*. Eles já convertem o *WSDL* em um arquivo *.pas* com todas as classes necessárias para consumir os serviços. *WSDL* é a sigla de *Web Services Description Language*, sendo este um padrão baseado em *XML* para descrever o serviço, contendo os métodos do *Web Service*. Funciona como uma espécie de *Type Library* do *Web Service*, além de ser usado para a validação das chamadas dos métodos.

O *WSDL* é uma especificação desenvolvida pelo *W3C* que permite descrever os *Web Services* segundo um formato *XML*, independentemente dos formatos de mensagem e dos protocolos de rede que sejam usados. O *WSDL* descreve os serviços disponibilizados à rede através de uma semântica *XML*, de forma que toda a informação necessária para se chamar um sistema distribuído e o procedimento necessário para que esta comunicação se estabeleça já estejam também descritos. Enquanto o *SOAP* especifica a comunicação entre um cliente e um servidor, o *WSDL* descreve os serviços oferecidos.

Definição do protocolo SOAP

SOAP (Simple Object Access Protocol) é um protocolo utilizado para invocar de forma remota um método qualquer informando seu endereço, o nome do método e os parâmetros para esse método quando necessário. Ele foi projetado para invocar aplicações remotas através de *RPC* (*Remote Procedure Calls* ou *Chamadas Remotas de Procedimento*) ou trocas de mensagens, em um ambiente independente de plataforma e linguagem de programação. Os dados enviados no protocolo *SOAP* são formatados em *XML* com determinadas regras e enviados normalmente por *HTTP*. Ele não define ou impõe

Nota do DevMan

B2B (Business to Business), é o nome dado ao comércio eletrônico associado a operações de compra e venda, de informações, de produtos e de serviços através da Internet ou através da utilização de redes privadas compartilhadas entre parceiros de negócios, substituindo assim os processos físicos que envolvem as transações comerciais. O B2B pode também ser definido como troca de mensagens estruturadas com outros parceiros comerciais a partir de redes privadas ou da Internet, para criar e transformar assim as suas relações de negócios.

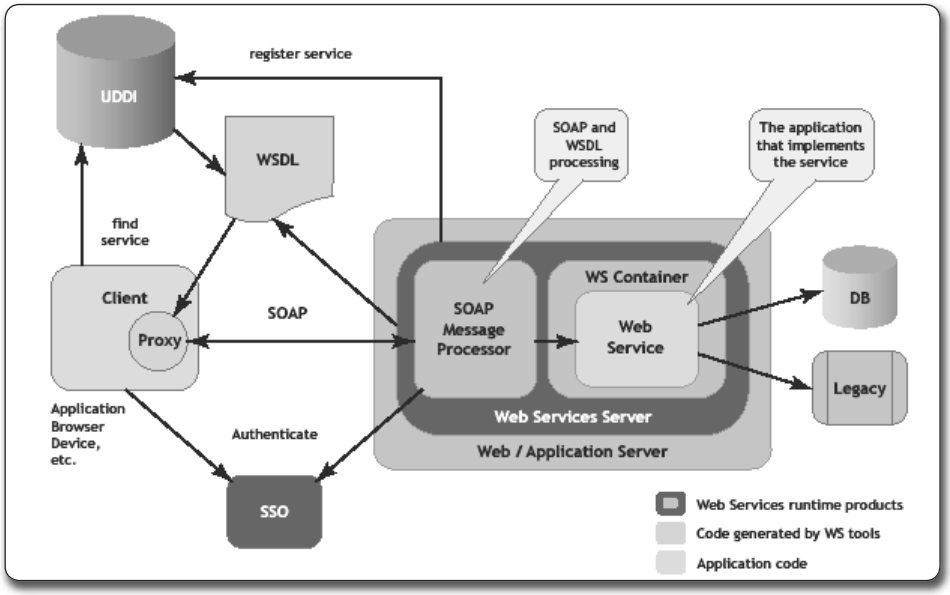


Figura 1. Diagrama da Arquitetura de um WS

qualquer regra, seja de programação ou de implementação. Este aspecto é extremamente importante, pois permite que, tanto o serviço quanto o cliente que invoca o serviço, sejam aplicações desenvolvidas sobre diferentes linguagens de programação.

Hoje o *SOAP* é o protocolo base dos *Web Services*, pois garante a interoperabilidade e intercomunicação entre diferentes sistemas, através da utilização da linguagem *XML* e do mecanismo de transporte *HTTP*. Por ser um protocolo de *RPC* que funciona sobre *HTTP*, o *SOAP* consegue de certa forma ultrapassar as restrições de segurança (*firewalls*) normalmente impostas a sistemas que utilizam *DCOM*. Em vez de usar *HTTP* para pedir uma página *HTML*, o *SOAP* envia uma mensagem de *XML* através do pedido *HTTP* e recebe uma resposta, através de *HTTP*. Observe na **Figura 2** como está dividida uma mensagem enviada utilizando protocolo *SOAP*.

Definições a parte, vamos partir agora para construção do nosso exemplo. Nele iremos criar nosso primeiro *Web Service* com algumas funções de exemplo, para que possam ser invocadas por diversas aplicações.

Nota do DevMan

W3C (World Wide Web Consortium) é um consórcio de empresas de tecnologia, atualmente com cerca de 500 membros. Fundado por Tim Berners-Lee em 1994 para levar a Web ao seu potencial máximo, por meio do desenvolvimento de protocolos comuns e fóruns abertos que promovem sua evolução e asseguram a sua interoperabilidade. O W3C desenvolve padrões para a criação e a interpretação dos conteúdos para a Web. Sites desenvolvidos segundo esses padrões podem ser acessados e visualizados por qualquer pessoa ou tecnologia, independente de hardware ou software utilizados, como celulares, PDAs, de maneira rápida e compatível com os novos padrões e tecnologias que possam surgir com a evolução da internet.

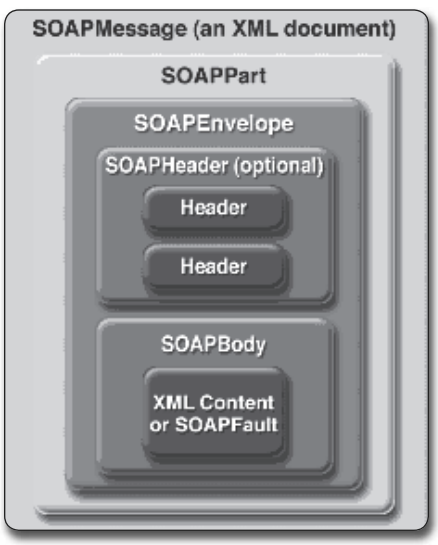


Figura 2. Exemplo de Mensagem SOAP

Preparando o Ambiente

Para nosso exemplos iremos usar o banco de dados *World Data Base* disponível no site do *MySql*. Para isso, acesse o link *http://dev.mysql.com/doc/#sampledb*. Na seção *Examples Data Base*, utilize o link para baixar a versão compactada deste banco de dados. Ao terminar o download descompacte o arquivo no diretório raiz. O arquivo *Word.sql* possui o *script* de criação da base de dados e toda a instrução de alimentação dos dados no mesmo. Este banco possui informações de diversos, senão todos, países do mundo, com as principais cidades relacionadas. Para se ter uma idéia são mais de 230 registros de países, 4000 registros de cidades e 900 línguas faladas ao redor do mundo. Isso é suficiente para criarmos serviços *Web* interessantes.

Abra o *prompt* do servidor *MySql* a após realizar o *login* execute o seguinte comando: *mysql> .\ C:/world.sql*. Isso irá

gerar o banco de dados *World* no seu servidor *MySql*. Você poderá usar a ferramenta de sua preferência para executar esta tarefa.

A idéia é que nós disponibilizemos alguns serviços através da nossa aplicação, por exemplo, informações sobre determinado país como língua oficial, presidente, população. Ou então as principais cidades de um país e assim por diante. Tudo isso ficará disponível através do nosso *Web Service* para ser consultado por qualquer aplicação, em qualquer linguagem e plataforma. Para termos uma noção geral dos dados com que podemos trabalhar a **Figura 3** mostra o *DER* do banco de dados *DBWorld*.

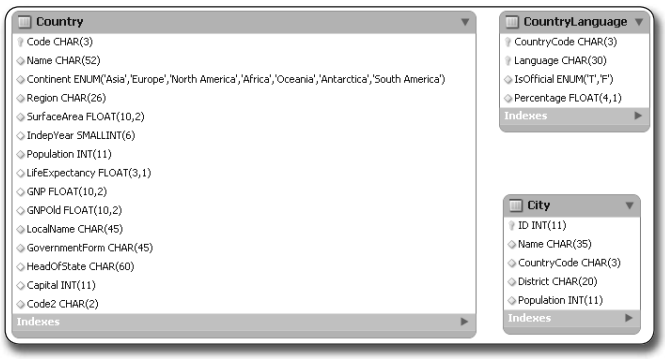


Figura 3. DER do banco de dados World

Banco de dados pronto e instalado vamos passar a criação da aplicação.

Criando a aplicação

Vamos iniciar nosso projeto criando uma nova aplicação no *Delphi for PHP*. Abra o *Delphi for PHP*, e acesse o menu *File>New>Aplication*. Quando criamos uma nova aplicação, o *D4PHP* já trás de graça um *Form*, geralmente chamado *Unit1*. Nós não precisaremos deste *Form*, até porque estamos criando um serviço, e sendo assim dispensamos qualquer interface. Agora temos um projeto sem *Form*, porém, precisamos de um container para nosso *Web Service*. Adicione para isso um *Data Module*. No menu *File>New>Data Module* conseguimos facilmente adicionar um *Data Module* a aplicação.

A vantagem de se usar o *D4PHP* para criar um *Web Service* é que ele trás embutido na *VCL* um componente que já encapsula as classes necessárias para se publicar serviços *Web*. Esse componente chama-se *Service*, bem sugestivo. Adicione ao *Data Module* um componente *Service*, localizado na paleta *Web Service*. Este componente é responsável por exportar os métodos que sua aplicação deseja prover. Basicamente o funcionamento deste componente é bem simples. Cada serviço prestado pela aplicação deve ser representado por um método, método este que deve ser global, ou seja, como estamos usando um *Data Module* os métodos devem estar fora da classe que representa o *Data Module*. Com os métodos devidamente definidos, o próximo passo é registrar estes métodos no *Web Service* através do componente *Service*. Simples assim!

ClubeDelphi PLUS www.devmedia.com.br/clubedelphi/portal.asp

No portal DevMedia você encontra um curso completo de MySql do mesmo autor, Rodrigo Carreiro, onde ele mostra todas as características do MySql, desde a instalação passando por procedures, triggers até backup e replicação.

http://www.devmedia.com.br/cursos/listcurso.asp?curso=136

Salve o *Data Module* como *WSDevMedia.php* e o projeto como *PrjWS.phprj*. Renomeie o *Data Module* para *DMWS* alterando a propriedade *Name* do mesmo. Da mesma forma, renomeie o componente *Service* para *ServiceDefault*. Para concluirmos esta primeira etapa, aproveite para renomear também a propriedade *Service Name*. Esta propriedade contém o nome do serviço que você irá publicar. Por padrão o nome do serviço é *VCL*. Altere para *Web Service World*.

Com isso temos nosso projeto devidamente criado e renomeado. Vamos criar nossa primeira função simplesmente para podermos conhecer os procedimentos necessários para registrar um serviço. Como não podia deixar passar, vamos ao famoso *Hello World*. Assim sendo na *unit* do *Data Module* vamos criar uma nova função. Não esqueça que esta função deve estar fora do escopo da classe *DMWS*. Assim sendo declare a função na *unit WSDevMedia.php* como visto a seguir:

```
function HelloWorld($mensagem){
   return ucwords($mensagem);
}
```

A função *Hello World* dispensa comentários. Nela apenas recebemos um parâmetro que conterá a mensagem a ser tratada, e na implementação da função apenas usamos o método *ucwords* para colocar a primeira letra de cada palavra da mensagem em letras maiúsculas. Até aí nenhuma novidade. Agora o que precisamos fazer é registrar esta função no nosso *Web Service* para que ela possa ser exportada. Selecione para isso o componente *ServiceDefault* e na paleta *Events*, localize e dê um duplo clique no evento *OnRegisterServices*. Isso irá criar uma função no nosso *Data Module*. O componente *Service* possui um método chamado *Register* que se encarrega do trabalho pesado para nós. No código a seguir temos o código que deverá ser adicionado no evento *OnRegisterServices* (**Listagem 1**).

Listagem 1. Código para registrar o método Hello World

```
1. $this->ServiceDefault->register('HelloWorld',
2. array('mensagem' => 'xsd:string'),
3. array('retorno' => 'xsd:string'),
4. 'http://localhost');
```

O método *Register* solicita quatro parâmetros principais, são eles: O nome do método que queremos registrar. Temos que atentar para o fato de o *PHP* ser *Case Sensitive*. Observe que o nome do método foi passado como as letras *H* e *W* em caixa alta. Com isso, tome o cuidado de verificar se o nome do método passado como parâmetro é idêntico ao nome da função declarada no *Script*. Na sequência, temos que passar os parâmetros de entrada e saída, **Linha 1** e **2**

respectivamente. Estes dois parâmetros devem ser passados como *array*, tendo seus itens passados com operador de associação. Como na nossa função temos apenas um parâmetro de entrada a **Linha 2** mostra exatamente a maneira correta de informar isso: à função register. Mensagem e identificador do parâmetro e *xsd:string* o tipo deste parâmetro passa no padrão *W3C*. Se por ventura tivessemos outros parâmetros a serem informados o código ficaria assim: *array ('mensagem' => 'xsd:string', 'param2' => 'xsd:string', 'param3' => 'xsd:string')*.

Na **Linha 3** temos um código similar a **Linha 2** porém, agora o registro que estamos fazendo é do parâmetro de saída. Como ficou definido, iremos retornar a mesma mensagem enviada, porém, agora formatada. Por este motivo temos apenas um parâmetro sendo registrado. A sintaxe é a mesma do parâmetro de entrada, caso tenhamos mais de um.

Pronto! Já temos nosso primeiro serviço pronto para ser consumido. Para testar temos que ativar o serviço. Para isso, altere a propriedade *Active* do componente *ServiceDefault* para *true* e execute a aplicação. Observe a **Figura 4**. Ela mostra a aplicação em execução com a interface padrão que o próprio componente *ServiceDefault* gerou.

Repare na presença do link *WSDL* na página que foi exibida. Este link nos submete ao conteúdo *XML* que descreve os serviços disponíveis. A classe *Service* já gera este descritor de serviços de forma automática dentro do padrão *W3C* para geração de *Web Service*. Basta enviar o link deste arquivo para seus clientes, para que eles importem e gerem os clientes *Web Service* na linguagem e plataforma que eles trabalham.

Criando a função para exportar DataSet

Como você deve ter reparado a função *HelloWorld* foi apenas para demonstrar como funciona o mecanismo de registro de métodos. A partir de agora vamos criar funções realmente úteis dentro do nosso contexto, que disponibilizam serviços de caráter geográfico, para que possamos extrair informações das nações ao redor da terra. Como essas informações estão no banco de dados, teremos que executar uma consulta na base para obter tais informações, até ai nenhum problema. O problema está em como exportar os dados que estão no nosso *DataSet*.

A resposta que será enviada ao cliente será sempre em pacotes *XML*, este é o padrão dos *Web Services*. Neste caso teremos que converter nosso *Result Set* em um conteúdo *XML* válido para ser transmitido via *Web Service*. Vamos então criar uma nova função da mesma maneira que criamos a função *HelloWorld*, fora do escopo do *DataModule*. Esta função não será registrada

Listagem 2. Função para converte Data Set em XML

```
1. function ExportToXML($select, $caption)
2. {
3.          $db = new Database;
4.          $db->Name = 'dbconnection';
5.          $db->DriverName = 'mysql';
6.          $db->DatabaseName = 'dbworld';
7.          $db->UserName = 'root';
8.          $db->UserPassword = 'root';
9.          $db->open();
10.
11.         $qry = new Query;
12.         $qry->Name = 'qryconsulta';
13.         $qry->Database = $db;
14.         $qry->SQL = $select;
15.         $qry->LimitCount = -1;
16.         $qry->open();
17.
18.         $doc = new DomDocument('1.0', 'UTF-8');
19.         $dados = $doc->createElement('Dados');
20.         $dados = $doc->appendChild($dados);
21.         while(!$qry->EOF)
22.         {
23.            $reg = $doc->createElement($caption);
24.            $reg = $dados->appendChild($reg);
25.            foreach($qry->Fields as $name => $value)
26.            {
27.               $field = $doc->createElement($name);
28.               $field = $reg->appendChild($field);
29.               $text = $doc->createTextNode($value);
30.               $text = $field->appendChild($text);
31.            }
32.            $qry->Next();
33.         }
34.         $xmlcontent = $doc->saveXML();
35.         return $xmlcontent;
36. }
```

Figura 4. Aplicação em execução

Nota do DevMan

O Document Object Model é uma API para documentos HTML e XML. Isto proporciona uma representação estrutural de um documento, habilitando você a modificar seu conteúdo e apresentação visual. O DOM foi desenhado para ser independente de qualquer linguagem de programação particular, fazendo a representação estrutural do documento disponível em uma simples, API consistente. O World Wide Web Consortium estabelece um padrão para o DOM, chamado o W3C DOM.

no *Web Service*, apenas servirá para converter o *ResultSet* em *XML*. A **Listagem 2** exibe o conteúdo da função.

Antes de comentar a **Listagem 2**, adicione à *uses* as *units use_unit("db.inc.php")* e *use_unit("dbtables.inc.php")*, para que seja permitido utilizar as classes *Database* e *Query* no *Data Module*. Observe que na **Linha 1** temos o cabeçalho da função, onde a mesma recebe dois parâmetros, o *select* que será executado no banco de dados, e um *Caption* que será usado para nomear o nó do registro no *XML*. Da **Linha 3** até a **Linha 8** temos os códigos necessários para criar um componente *DataBase*. Este componente será responsável por realizar a conexão com nossa base de dados. Repare nos parâmetros passados, não se esqueça de alterá-los de acordo com os dados de seu banco de dados. Atente para a propriedade *DataBaseName*. Eu inseri o valor *DBWorld*, pois este é o nome da minha base de dados. Não se esqueça de configurá-la corretamente.

Nas **Linhas 11** a **15** temos o código que utilizamos para criar um componente *Query*. Este componente será responsável por executar o *select* passado como parâmetro, no banco de dados. Atente para o detalhe de que passamos o componente *$db* para a propriedade *database*. Na **Linha 18** criamos um objeto *DomDocument*. Este objeto utilizamos o padrão *DOM* para criar em memória uma árvore com os nós formando a estrutura do nosso documento *XML*. Todo documento *XML* deve ter um único elemento raiz e este está sendo criado na **Linha 19**. A **Linha 20** cria um sub-nó abaixo do nó raiz para receber os *fields* presentes no nosso *ResultSet*.

Na **Linha 21** iniciamos nosso *loop* para percorrermos cada registro do resultado na nossa consulta, e para cada registro localizado nós criamos mais um elemento (**Linha 23)**, e este por sua vez, tem o parâmetro *$caption* passado como seu identificador ou *label*. Na **Linha 24** criamos mais um nó, agora como um sub-nó do registro. Mais um *loop* é iniciado, agora para percorrer todas as colunas retornadas no *select* e para cada coluna criamos um novo elemento (**Linha 27**), e atribuímos o nome do *Field* a ele. Ainda no mesmo laço criamos um nó do tipo *TextNode* para receber o conteúdo do *Field*. Na **Linha 34** invocamos o método que irá gerar o arquivo *XML* propriamente dito. Este arquivo é retornado através da função na **Linha 35**. A **Listagem 3** exibe um exemplo de como ficará a estrutura se estivéssemos exportado os dados de um país por exemplo.

Listagem 3. Exemplo de estrutura XML gerada

```
<?xml version="1.0" encoding="UTF-8"?>
<Dados>
 <pais>
  <Code>AFG</Code>
  <Name>Afghanistan</Name>
  <Continent>Asia</Continent>
  <Region>Southern and Central Asia</Region>
  <SurfaceArea>652090.00</SurfaceArea>
  <IndepYear>1919</IndepYear>
  <Population>22720000</Population>
  <LifeExpectancy>45.9</LifeExpectancy>
  <GNP>5976.00</GNP>
  <GNPOld></GNPOld>
  <LocalName>Afganistan/Afqanestan</LocalName>
  <GovernmentForm>Islamic Emirate</GovernmentForm>
  <HeadOfState>Mohammad Omar</HeadOfState>
  <Capital>1</Capital>
  <Code2>AF</Code2>
 </pais>
</Dados>
```

Listagem 4. Métodos para exportação

```
function CountryByContinent($Continent){
   return ExportToXML("Select * from Country
     where continent = '$Continent'", 'Countries');
}
function LanguagesByCountry($Country){
   return ExportToXML("SELECT Language, percentage
      FROM countrylanguage L, country c
      where l.Countrycode = c.Code
      and c.Name = '$Country'
      order by Language", 'Languages');
}
function GetSimpleCountry($Country){
   return ExportToXML("SELECT c.`Name`, ct.`name` , c.`Continent`,
      c.`Population`, c.`SurfaceArea`
      FROM country c, city ct
      where c.capital = ct.id
      and c.name = '$Country' ", 'Pais');
}
```

Listagem 5. Registrando os métodos

```
$this->ServiceDefault->register('CountryByContinent',
      array('Continent' => 'xsd:string'),
      array('Countries' => 'xsd:string'),
      'http://localhost');

$this->ServiceDefault->register('LanguagesByCountry',
      array('Country' => 'xsd:string'),
      array('Languages' => 'xsd:string'),
      'http://localhost');

$this->ServiceDefault->register('GetSimpleCountry',
      array('Country' => 'xsd:string'),
      array('Information' => 'xsd:string'),
      'http://localhost');
```

Nosso método está pronto para exportar qualquer consulta realizada à base de dados. Passemos agora a criação dos métodos que farão uso desta rotina e que serão exportados.

Criando e exportando os métodos

Sem muito papo vamos direto para a criação das funções. Assim como criamos os demais métodos fora do escopo do *Data Module*, vamos criar mais 3 para que possamos concluir nossa aplicação. A **Listagem 4** descreve as funções que devem ser declaradas.

O código da **Listagem 4** dispensa comentários. Observe que apenas invocamos o método *ExportToXML* passando ao mesmo a instrução *SQL* a ser executada no banco de dados. O retorno deste *select* é repassado para quem invocou o método. Métodos criados, passemos então a fase onde iremos registrar estas funções no *Web Service*. O código da **Listagem 5** se encarrega disto.

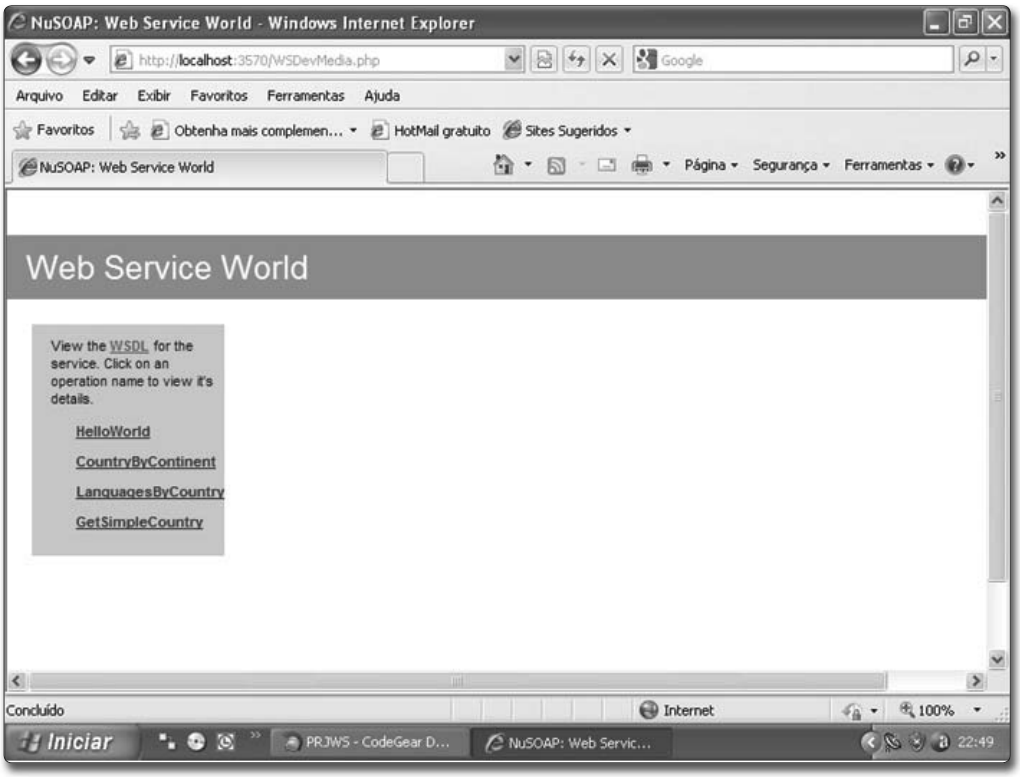


Figura 5. Métodos publicados no Web Service

Repare que o procedimento foi o mesmo utilizado para registrar o método *HelloWorld*. Não se esqueça de atentar para o nome do método. Lembre-se que o *PHP* é *Case Sensitive*, portanto, uma letra minúscula que seja, pode impedir que a aplicação funcione corretamente.

Execute a aplicação e repare que agora temos, além do método *HelloWorld*, mais três métodos disponíveis para serem consumidos conforme mostra a **Figura 5.**

Conclusão

Com isso encerramos aqui nosso primeiro *Web Service*. Gostaria de lembrá-los que em um processo que envolve *Web Services* teremos sempre os dois lados da moeda. A aplicação servidora, que publica os serviços que estão sendo prestados e a aplicação cliente, que irá consumir estes serviços. Neste artigo abordamos apenas a primeira parte. No próximo artigo trataremos da criação dos clientes *Web Services*. Faremos 3 aplicações: uma em *Win32*, uma em *ASP.NET* e uma em *PHP*. Isso para mostrar o quanto utilizar *Web Services* pode nos ajudar a integrar nossos sistemas com o mundo exterior.

Um abraço a todos e até a próxima! ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Nesta seção você encontra artigos sobre a linguagem PHP e a ferramenta Delphi for PHP

Introdução a orientação a objetos com PHP

Conheça os princípios básicos da POO com PHP

Resumo do DevMan

Programação orietada a objetos gera polêmica em toda e qualquer linguagem de programação, isso é certo. Mas não podemos deixar de notar que é uma das melhores práticas de desenvolvimento de modo geral. Certamente desenvolver utilizando POO, padrões de projeto e outras práticas que já estamos acostumados a ouvir, é satisfatório e agiliza os processos de desenvolvimento.

Nesse artigo veremos um pouco de POO com PHP puro e saber como aplicar os principais conceitos nessa linguagem.

Nesse artigo veremos

- Criação de Classes;
- O que são Atributos;
- Como criar métodos;
- Introdução a Herança.

Qual a finalidade

Apresentação dos conceitos básicos da POO no PHP.

Quais situações utilizam esse recurso?

Desenvolver em orientação a objetos é muito importante, tanto na produtividade como na reutilização de seus códigos. Nesse artigo veremos uma introdução rápida aos conceitos de POO com PHP puro.

Rafael Silva

rafael@rlsystem.com.br

Trabalha com análise e desenvolvimento de sistemas. Atua na área de desenvolvimento desktop utilizando Delphi, C#, e foco na web com ASP.NET, PHP, CSS, AJAX, sempre utilizando os padrões da Web 2.0 em seus projetos. Utiliza banco de dados PostGreSQL, MySQL, SQL Server e Firebird.

Programação a orientação a objetos é uma metodologia muito importante hoje em dia. Assim como em outras linguagens, no *PHP* também é possível, apesar da maioria dos desenvolvedores ainda não utilizarem ela na prática. Na minha opinião, desenvolver em modo procedural pode ser um desenvolvimento mais rápido, mas com certeza a manutenção do seu sistema vai ser muito mais complicada.

Entender *POO* é fácil, vejamos um exemplo: quando temos qualquer aplicação para desenvolver e resolvemos criá-la em *POO* ganhamos diversas vantagens nisso. Uma delas é que cada membro da equipe pode se incubido a desenvolver determinadas partes de tal aplicação. Em outras palavras, um portal de notícias por exemplo, faria com que um programador fosse responsável pela classe de notícias, outro a classe

de usuários, e assim sucessivavemente. No final das contas, cada um apenas chamaria as partes envolvidadas no site via sua interface *Web*.

É lógico que esse é um exemplo simples, mas podemos refletir em cima disso e perceber o quanto ganhamos em produtividade. Outra vantagem é que podemos alterar somente partes do sistema, ou seja, quando precisamos implementar ou corrigir um erro em determinada parte dosistema, basta alterarmos somente as classes que causam o problema ou que precisam sofrer insersão de funcionalidades. Resumindo: Altere a classe e pronto.

Se procurarmos na internet sobre Programação Procedural encontraremos diversos links que apontam para o real significado. No site *Wikiédia* encontramos a seguinte definição:

"O termo Programação Procedural (ou programação procedimental) é as vezes utilizado como sinônimo de Programação Imperativa (Paradigma de programação que especifica os passos que um programa deve seguir para alcançar um estado desejado), mas o termo pode se referir à um paradigma de programação baseado no conceito de chamadas a procedimento. Procedimentos, também conhecidos como rotinas, subrotinas, métodos, ou funções (que não devem ser confundidas com funções matemáticas, mas são similares àquelas usadas na programação funcional) simplesmente contém um conjunto de passos computacionais a serem executados. Um dado procedimento pode ser chamado a qualquer hora durante a execução de um programa, inclusive por outros procedimentos ou por si mesmo."

Já a programação procedural, todo o código acaba ficando misturado e isso dificulta até a depuração do sistema para encontrar erros. Um dos principais erros do progamador que desenvolve em modo procedural, é escrever instrução *SQL* em telas. Isso também é uma prática pouco aconselhada, pois prejudica a performance da aplicação e dificulta a manutenção de todo o sistema. Mas isso é assunto para outro artigo.

Veja como é mais trabalhoso e por experiência própria sempre tive esse problema. No começo pode ser difícil você entendermos os conceitos, mas saliento que vale muito a pena. Com essa nossa introdução vamos criar duas classes simples que nada mais faz do que mostrar uma transação bancária.

O objetivo desse artigo não é construírmos uma aplicação complexa e detalhada, e sim entender e saber os principais conceitos para *POO* de maneira simples no *PHP*.

Entendendo e criando classes

As classes são utilizadas para encapsular funcionalidades do sistema, permitir a reutilização de código e com isso agilizar o processo de desenvolvimento de qualquer aplicação. Um exemplo claro disso é acesso a banco de dados. Quando precisamos acessar um banco e listar informações de determinadas tabelas, temos que seguir uma estrutura básica de desenvolvimento, veja um exemplo simples:

```
<?php
$conexao = mysql_connect('localhost', 'usuario', 'senha');
mysql_db_select('meu_banco_de_dados');
?>
```

O que acabamos de ver pode ser repetir diversas vezes em uma aplicação PHP durante o ciclo do site. Se criarmos uma classe para encapsular isso, podemos apenas criar funções de conexão e desconexão do banco e utilizá-las quando necessário, resumidamente:

```
<?php
## Listar clientes
conectar_ao_banco;

## Imprimir consulta
## códigos para impressão dos dados em tela
?>
```

Perceba que estamos chamando uma função denominada *conectar_ao_banco()*. É claro que esse é um exemplo bastante simples, mas totalmente funional. Fazendo isso, estamos reutilizando os códigos que fazem a conexão com o banco de dados.

Para criar uma classe em PHP, usamos a palavra reservada *class* seguida do nome da mesma. Lembrando apenas que não é permitido usar números no início do nome da classe e nem caracteres especiais, além de não usar palavras reservadas para tal propósito.

Nota

Caso tenha dúvidas quanto as palavras reservadas no PHP, acesse o link www.php.net/manual/pt_BR/reserved.keywords.php e visualize todas as palavras reservadas e seus respectivos significados no PHP.

Vamos ver como criar uma classe em *PHP*. Abra o *Bloco de Notas* do *Windows* ou qualquer outro editor de sua preferência para editar nossa classe. Digite um código semelhante a **Listagem 1**. Veja que criamos a classe com o nome *Conta* e outra classe denominada *Pessoa*, respectivamente **.** Salve o arquivo como *classes.php*.

Listagem 1. Classe Conta e Pessoa

```
<?php
class Conta{
  ## Código da Classe Conta
  echo "Você chamou a classe Conta."
}
?>

class Pessoa{
  ## Código da Classe Pessoa
  echo "Você chamou a classe Pessoa."
}
?>
```

Não há segredos. O arquivo salvo *classes.php* possui duas classes que podem ser chamadas por nossas aplicações *PHP*. Dentro de cada classe podemos criar *n* funções, que chamamos de métodos. Veremos isso mais adiante.

O que são atributos?

Atributos são aspéctos de nosso objeto, ou seja, no nosso caso os objetos são *Conta* e *Pessoa*. Outra explicação para atributos é que são características de tais objetos. Fazendo uma analogia, pensemos em calçados. Calçados possuem características e comum, por exemplo: tênis costumam ter cadarços. Cadarço

nesse caso é um atributo do objeto tênis, logo podemos acessar esse atributo usando algo como:

```
tenis.cadarço = 'qualquer-coisa';
```

É importante estar ciente que um atributo, assim como um método pode ser privado (*Private*), público (*Public*) ou protegido (*Protected*), onde:

• *Public:* é acessível de qualquer classe, subclasse ou em qualquer parte do seu script;
• *Private:* é acessível somente pela sua classe, ficando restrito a subclasses ou a qualquer parte de script que a utiliza;
• *Protected:* é acessível somente a sua classe ou subclasses.

Para declaramos um atributo é simples, primeiro sua visibilidade depois seu nome. Nosso objeto *Contas* precisa basicamente de número e saldo, enquanto o objeto *Pessoa* precisa de somente *Nome*. Sim colocamos somente alguns atributos básicos para que nossa classe tenha sentido. Veja a **Listagem 2** que criamos dois atributos na classe *Conta* e um na classe *Pessoa*.

Listagem 2. Atributos da classe Conta e Pessoa

```
<?php
class Conta{
        public $numero;
        public $saldo;
}
?>
class Pessoa{
        public $nome;
}
?>
```

Como mencionei na analogia, é possível acessar esses atributos usando uma instrução simples, respeitando é claro o conceito de visibilidade dos atributos.

```
Conta.numero;
Conta.saldo;
Pessoa.nome;
```

Métodos

Métodos são ações de um objeto, de forma clara é objetiva posso dizer que o objeto *Conta* pode ter métodos de *Sacar*, *Depositar* e o objeto *Pessoa SetarNome* que atribui o valor a uma propriedade no nosso caso. Voltando a nossa analogia, podemos imaginar algumas ações do calçado, como *Amarrar, Desamarrar, Calçar* etc. Isso significa que podemos codificar *ordens* para serem executadas quando essa ações acontecerem. Podemos chamar essas ações também de eventos.

Da mesma forma que um atributo, métodos também tem possui o conceito de visibilidade (*public, private, protected*). Para criarmos um método, declaramos sua visibilidade e depois à palavra reservada *function* seguida de seu nome.

Observe que quando vamos chamar algum método ou atributo dentro do contexto da nossa classe, utilizamos a palavra reservada *$this*. Veja na **Listagem 3** nossos métodos.

Na **Listagem 3** temos listadas os códigos-fonte dos métodos de Conta e Pessoa. Nas **Linhas 3** e **4** seguem as declarações dos atributos de Conta e logo em seguida a função (*método*)

Listagem 3. Métodos da classe Conta e Pessoa

```
01 <?php
02 class Conta{
03      public $numero;
04      public $saldo;
05      public function Sacar($valor){
06         if ($valor > $this->saldo){
07            return false;
08         }else{
09            $this->saldo -= $valor;
10            return true;
11         }
12      }
13      public function Deposita($valor){
14         $this->saldo += $valor;
15      }
16 }
17
18 class Pessoa{
19      public $nome;
20      public function SetarNome($nome){
21         return $this->nome = $nome;
22      }
23 }
24 ?>
```

Sacar(). Note que recebemos como parâmetro o valor do saque (*$valor*). Se o valor for maior que o saldo na conta, retornamos falso para evitar negativar a conta. Caso contrário, subtraímos o valor do saldo e retornamos verdadeiro indicando que o saque foi efetuado.

Poucas linhas abaixo declaramos o método Deposita(), que apenas incrementa no saldo o valor recebido. É claro que novamente temos um exemplo bastante simples. Se fosse um sistema bancário, certamento teríamos que fazer outras checagens, como por exemplo saber que o correntista possui limite bancário para utilização.

A classe *Pessoa* possui um atributo chamado *nome* e um método chamado *SetarNome();* No método, recebemos o nome que deve ser usado. Então, configuramos o atributo nome com o conteúdo do parâmetro recebido.

Veja que é bem simples trabalhar dessa forma no PHP e ainda estamos preparando o sistema para reutilização de código.

Heranças

A herança em programação orietada a objetos segue basicamente a mesma lógica da vida real. Quando herdamos de uma outra classe, estamos trazendo junto seus métodos, atributos e tudo que ela possuir. Isso é mais uma vantagem da *POO*, estamos reutilizando mais uma vez um código-fonte. Se fizermos novamente a analogia do calçado podemos dizer que na classe calçado há vários tipos de objetos. Tênis, sapatos, sandálias, etc. Todos eles possui características entre si, por exemplo: solado.

A partir da herança, podemos criar quando objetos quisermos. Se criarmos por exemplo uma classe *Tênis* herdada de *Calçado*, saberemos que ela possuirá os atributo *Solado* e os métodos *Amarrar* e *Desamarrar*. Nada impede agora de criamos outros métodos e atributos, como por exemplo *Cor* como atributo e os métodos *Calçar/Descalçar*.

Em outras palavras, como o próprio nome diz, herança é herdar de alguém ou de alguma coisa algo, assim como na vida real. No nosso caso temos que fazer a classe *Conta* herdar de *Pessoa*, e com isso obter todos seus métodos e atributos. Para

realizar isso é muito simples, basta acrescentar logo após o nome da classe a palavra reservada *extends* e o nome da classe que vamos herdar. Caso a classe a ser herdada esteja em um arquivo PHP separado é necessário invocarmos ese arquivo usando o método *PHP require_once* (**Listagem 4**).

Listagem 4. Classe Conta com herança

```
01 <?php
02 require_once("Pessoa.php");
03 class Conta extends Pessoa{
04      public $numero;
05      public $saldo;
06      public function Sacar($valor){
07         if ($valor > $this->saldo){
08            return false;
09         }else{
10            $this->saldo -= $valor;
11            return true;
12         }
13      }
14      public function Deposita($valor){
15         $this->saldo += $valor;
16      }
17 }
18 ?>
```

Na **Listagem 4** eu separarei a classe Pessoa enviando-a para outro arquivo chamado *Pessoa.php* propositalmente para que possamos ver como importar uma classe de outro arquivo. Usamos o método *require_onde()* como mencionado anteriormente. Na sequencia alteramos a classe *Conta* herdando de *Pessoa* usando a palavra reservada *extends* (**Linha 3**). Os atributos continuam como antes assim como os métodos.

Instância

Instância, grosseiramente falando, é uma cópia do objeto em memória para que possamos utilizá-l, exceto objetos estáticos. Cada vez que criamos/invocamos um objeto, uma instância dele é criada.

Para exemplificar isso, crie um novo arquivo PHP onde faremos uso de nossas classes. Definimos que a variável *$obj* será a instância da nossa classe. Para isso você apenas atribui a essa variável a palavra reservada *new* e o nome da nossa classe (*objeto*). Veja na **Listagem 5** como fazemos isso. Com isso temos acesso a todos os atributos e métodos desse *objeto*. Conforme a **Figura 1**, veja o resultado do nosso *script*.

Na *Linha 2* usando novamente o método *require_onde()* para que possamos ter acesso as classes incluídas nesse arquivo. Em seguida criamos a classe *Conta()*, ou seja, a instanciamos. Invocamos o método *SetarNome()* passando como parâmetro fixo o nome *Rafael Silva*. Chamamos o método *Deposita()* e *Sacar()* nas **Linhas 5** e **6**. E por fim imprimimos o resultado da função em tela usando *echo*.

Figura 1. Resultado do script

Listagem 5. Chamada da Classe

```
01 <?php
02 require_once 'Conta.php';
03 $obj = new Conta();
04 $obj->SetarNome("Rafael Silva");
05 $obj->Deposita(600);
06 $obj->Sacar(100);
07 echo $obj->nome . " possui o valor de: R$ " . $obj->saldo;
08 ?>
```

Até aqui vimos de maneira bastante simples como trabalhar com *POO* em aplicações *PHP* demostrando as principais noções de orientação a objetos. Mas é claro que ainda é preciso entrar em conceitos e outros fatores ainda mais profundos para entendermos perfeitamente o assunto. *POO* é assunto para centenas e centenas de artigos e em qualquer linguagem.

Em outros artigos, faremos a criação completa de um sistema de contas a pagar e cobrança utilizando o PHP como principal linguagem de programação. Até a próxima.

Conclusão

Posso citar que a orientação a objetos se encaixa muito bem com aplicações que utilizam banco de dados, já que podemos desenvolver nossos sistemas totalmente em camadas separando persistência de dados com visualização e assim por diante. Assim vamos ter uma aplicação muito mais eficiente e de fácil manutenção.

Adquirindo o conceito e entendendo perfeitamente a POO, independentemente da linguagem, com certeza o fará ter alto ganho de produtividade além de maximizar as possibilidades de seus sistemas e projetos. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Existem momentos onde a estrutura faz toda a diferença.
Se o tempo é ruim, você encontra na Alog um porto seguro. Isto porque sua empresa reduz custos e melhora a governança em TI. Entre em contato e descubra porque a Alog é líder em hosting gerenciado.
REDUÇÃO DOS CUSTOS
AUMENTO DA GOVERNANÇA EM TI
0800 282 3330 | alog.com.br
ALOG
DATA CENTERS DO BRASIL
HOSTING GERENCIADO | COLOCATION | CONTINGÊNCIA | E-MAIL CORPORATIVO | CONECTIVIDADE | SERVIÇOS PROFISSIONAIS

VOCÊ levou anos para desenvolver, testar e colocar seu produto no mercado e **ELE** apenas alguns minutos para **PIRATEÁ-LO.**

SafeNet tem à sua disposição a proteção de software mais avançada e segura. As Chaves de Hardware Sentinel oferecem tecnologia que incorpora criptografia de chave pública, criptografia AES e autenticação interna para garantir o mais alto nível de proteção anti-pirataria.

Peça seu Kit de Desenvolvimento hoje mesmo.

Mais informações www.safenet-inc.com/antipirataria/01
e receba o Estudo de Caso.

SafeNet®
The Foundation of Information Security